

*** USER'S MANUAL ***

FCC ID : XOJGA1000

The Federal Communication Commission Statement

This equipment has been tested and found to comply with the limits for a Class B Digital Device, pursuant to Part 15 of the FCC rules. These limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates, uses and can radiate radio frequency energy and, if not installed and used in accordance with the instruction, may cause harmful interference to radio communication. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause harmful interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one of more of the following measures: -

- Reorient or relocate the receiving antenna.
- Increase the separation between the equipment and receiver.
- Connect the equipment into an outlet on a circuit different from that to which the receiver is connected.
- Consult the dealer or an experienced Radio/TV technician for help.

Use only shielded cables to connect I/O devices to this equipment. You are cautioned that change or modifications not expressly approved by the party responsible for compliance could void your authority to operate the equipment.

THIS DEVICE COMPLIES WITH PART 15 OF FCC RULES. OPERATION IS SUBJECT TO THE FOLLOWING TWO CONDITIONS: -

1. This device may not cause harmful interference and
2. This device must accept any interference received, including interference that may cause undesired operation.

The antenna used for this transmitter must not be collocated or operation in conjunction with any other antenna or transmitter.

Table of Contents

Taiko R2	1
Legal Information	1
Overview	4
Our Language Philosophy	4
System Components	7
Objects	8
Events	8
Getting Started	9
Preparing Your Hardware	9
Starting a New Project	10
Writing Code	11
Building, Uploading and Running	14
Compiling a Final Binary	15
Programming with TIDE	15
Managing Projects	15
The Structure of a Project	15
Creating, Opening and Saving Projects	17
Templates	17
Adding, Removing and Saving Files	18
Resource Files	20
Built-in Image Editor	20
Coding Your Project	22
Project Browser	22
Code Auto-completion	23
Code Hinting	24
Tooltips	24
Supported HTML Tags	26
Making, Uploading and Running an Executable Binary	26
Two Modes of Target Execution.....	27
Debugging Your Project	28
Target States	28
Exceptions	29
Program Pointer	30
Breakpoints	30
The Call Stack and Stack Pointer.....	31
Stepping	32
The Watch	33
Scopes in Watch	37
Code Profiling	37
Project Settings	38
Programming Fundamentals	39
Program Structure	39
Code Basics	40
Naming Conventions	42
Introduction to Variables, Constants and Scopes	43
Variables And Their Types	43

Type Conversion	45
Type conversion in expressions.....	48
Compile-time Calculations	49
Arrays	50
Structures	54
Enumeration Types	55
Understanding the Scope of Variables.....	57
Declaring Variables	60
Constants	60
Introduction to Procedures	62
Passing Arguments to Procedures.....	64
Memory Allocation for Procedures	66
Introduction to Control Structures	67
Decision Structures	67
Loop Structures	68
Doevents	68
Using Preprocessor	71
Scope of Preprocessor Directives.....	73
Working with HTML	74
Embedding Code Within an HTML File.....	76
Understanding Platforms	77
Objects, Events and Platform Functions.....	78

Language Reference 78

Statements	79
Const Statement	79
Declare Statement	79
Dim Statement	81
Doevents Statement	82
Do... Loop Statement	82
Enum Statement	84
Exit Statement	85
For... Next Statement	86
Function Statement	87
Goto Statement	88
If.... Then.... Else Statement	89
Include Statement	90
Select-Case Statement	91
Sub Statement	93
Type Statement	94
While-Wend Statement	95
Keywords	96
As	96
Boolean	97
ByRef	97
Byte	97
ByVal	97
Char	97
Else	97
End	97
False	98
For	98
Integer	98
Next	98
Public	98
Short	98
Step	98

String	98
Then	98
Type	99
To	99
True	99
Word	99
Operators	99
Error Messages	101
C1001	101
C1002	101
C1003	102
C1004	102
C1005	102
C1006	103
C1007	103
C1008	104
C1009	104
C1010	104
C1011	105
C1012	105
C1013	106
C1014	106
C1015	106
C1016	107
C1017	107
C1018	107
C1019	108
C1020	108
C1021	108
C1022	109
C1023	109
C1024	110
L1001	110
L1002	110
L1003	111
L1004	111
L1005	111
L1006	111
L1007	112
L1008	112
L1009	112
Objects, Properties, Methods, Events	113
Development Environment	113
Installation Requirements	113
User Interface	113
Main Window	114
Operation Modes	114
Menu Bar	115
File Menu	115
Edit Menu	116
View Menu	117
Project Menu	117
Debug Menu	118
Image Menu	119
Window Menu	119

Help Menu	120
Toolbars	120
Project Toolbar	120
Debug Toolbar	121
Image Editor Toolbar	122
Tool Properties Toolbar	122
Selection Tool Properties	123
Paint Tool Properties	123
Eraser Tool Properties	123
Text Tool Properties	123
Line Tool Properties	124
Rectangle Tool Properties	124
Ellipse Tool Properties	125
Zoom Tool Properties	126
Status Bar	126
Dialogs	126
Project Settings	127
New Project	127
Add File to Project	128
Graphic File Properties Dialog	128
Panes	128
Call Stack	128
Output	129
Project	129
Browser	129
Files	130
Watch	130
Colors	130
Language Element Icons	131

Glossary of Terms 131

Compilation Unit	131
Compiler	131
Construct	132
Cross-Debugging	132
Identifier	132
Keyword	132
Label	132
Linker	132
P-Code	132
Syscall	132
Target	133
Virtual Machine	133

Platforms 133

Platform Specifications	133
EM202 Platform	134
Memory Space	134
Supported Variable Types	134
Supported Functions (Syscalls)	134
Supported Objects	135
Platform-dependent Constants	135
Enum pl_redir	136

Enum pl_io_num	136
Platform-dependent Programming Information.....	136
EM1000 and EM1000W Platforms	139
Memory Space	140
Supported Objects	140
Platform-dependent Constants.....	140
Enum pl_redir	141
Enum pl_io_num	142
Enum pl_int_num	144
Enum pl_sock_interfaces	145
Platform-dependent Programming Information.....	145
EM1202 and EM1202W Platforms	149
Memory Space	149
Supported Objects	150
Platform-dependent Constants.....	150
Enum pl_redir	150
Enum pl_io_num	151
Enum pl_int_num	153
Enum pl_sock_interfaces	154
Platform-dependent Programming Information.....	154
EM1206 and EM1206W Platforms	158
Memory Space	158
Supported Objects	159
Platform-dependent Constants.....	159
Enum pl_redir	159
Enum pl_io_num	160
Enum pl_int_num	162
Enum pl_sock_interfaces	162
Platform-dependent Programming Information.....	163
DS1202 Platform	166
Memory Space	167
Supported Objects	167
Platform-dependent Constants.....	167
Enum pl_redir	167
Enum pl_io_num	169
Enum pl_int_num	170
Enum pl_sock_interfaces	170
Platform-dependent Programming Information.....	171
DS1206 Platform	174
Memory Space	175
Supported Objects	175
Platform-dependent Constants.....	175
Enum pl_redir	175
Enum pl_io_num	177
Enum pl_int_num	178
Enum pl_sock_interfaces	179
Platform-dependent Programming Information.....	179
Common Information	182
Supported Variable Types (T1000-based Devices).....	183
Supported Functions (T1000-based Devices).....	183
LED Signals	184
Debug Communications	185
Project Settings Dialog	186
Device Explorer	187
Function Reference	189
Asc Function	189
Bin Function	190
Cfloat Function	190

Chr Function	191
Date Function	191
Daycount Function	192
Ddstr Function	192
Ddval Function	193
Ftostr Function	194
Hex Function	195
Hours Function	195
.Insert Function	196
Instr Function	197
Lbin Function	197
Left Function	198
Len Function	198
Lhex Function	199
Lstr Function	199
Lstri Function	200
Lval Function	200
Md5 Function	201
Mid Function	202
Mincount Function	203
Minutes Function	203
Month Function	204
Random Function	205
Right Function	205
Sha1 Function	205
Str Function	207
Strgen Function	207
Stri Function	208
Strsum Function	209
Strtov Function	209
Val Function	210
Vali Function	210
Weekday Function	211
Year Function	211
Object Reference	212
Sys Object	212
Overview	212
On_sys_init Event	212
Buffer Management	213
System Timer	214
PLL Management	215
Serial Number	216
Miscellaneous	217
Properties, Methods, Events	217
.Buffalloc Method	217
.Currentpll R/O Property (Selected Platforms Only)	218
.Freebuffpages R/O Property	218
.Halt Method	219
.Newpll Method (Selected Platforms Only)	219
On_sys_init Event	220
On_sys_timer Event	220
.Onsystimerperiod Property (Selected Platforms Only)	220
.Reboot Method	221
.Runmode R/O Property	221
.Serialnum R/O Property	221
.Setserialnum Method	222
.Resettype R/O Property	222
.Timercount R/O Property	223

.Totalbuffpages R/O Property.....	223
.Version R/O Property	223
Ser Object	224
What's new in V1.1	224
Overview	225
Anatomy of a Serial Port	225
Three Modes of the Serial Port.....	225
UART Mode	226
Wiegand Mode	229
Clock/Data Mode	232
Port Selection	234
Serial Settings	236
Sending and Receiving Data (TX and RX buffers).....	239
Allocating Memory for Buffers.....	239
Using Buffers	240
Buffer Memory Status	240
Receiving Data	241
Sending Data	243
Handling Buffer Overruns	244
Redirecting Buffers	245
Sinking Data	245
Properties, Methods, Events.....	246
.Autoclose Property	248
.Baudrate Property	248
.Bits Property	249
.Ctsmap property (Selected Platforms Only).....	249
.Dircontrol Property	250
.Div9600 R/O Property	250
.Enabled Property	251
.Escchar Property	251
.Estype Property	251
.Flowcontrol Property	253
.Getdata Method	253
.Interchardelay Property	254
.Interface Property	255
.Mode Property	255
.Newtxlen R/O Property	256
.Notifysent Method	257
.Num Property	257
.Numofports R/O Property.....	258
.On_ser_data_arrival Event.....	258
.On_ser_data_sent Event	258
.On_ser_esc Event	259
.On_ser_overrun Event	259
.Parity Property	260
.Redir Method	260
.Rtsmap Property (Selected Platforms Only).....	261
.Rxbuffrq Method	262
.Rxbuffsize R/O Property	262
.Rxclear Method	263
.Rxlen R/O Property	263
.Send Method	264
.Setdata Method	264
.Sinkdata Property	265
.Txbuffrq Method	265
.Txbuffsize R/O Property	266
.Txclear Method	266
.Txfree R/O Property	266
.Txlen R/O Property	267

Net Object	267
Overview	268
Main Parameters	268
Checking Ethernet Status	269
Properties, Methods, Events	269
.Mac R/O Property	269
.Ip Property	270
.Netmask Property	270
.Gatewayip Property	270
.Failure R/O Property	271
.Linkstate R/O Property	271
On_net_link_change Event	271
On_net_overrun Event	272
Button Object	272
On_button_pressed Event	273
On_button_released Event	273
.Pressed R/O Property	274
.Time R/O Property	274
Sock Object	274
Overview	275
Anatomy of a Socket	276
Socket Selection	276
Handling Network Connections	277
TCP connection basics	278
UDP "connection" basics	278
Accepting Incoming Connections	279
Accepting UDP broadcasts	281
Understanding TCP Reconnects	281
Understanding UDP Reconnects and Port Switchover	283
Incoming Connections on Multiple Sockets	286
Establishing Outgoing Connections	287
Sending UDP broadcasts	288
Closing Connections	290
Checking Connection Status	292
More On the Socket's Asynchronous Nature	294
Sending and Receiving data	297
Allocating Memory for Buffers	297
Using Buffers in TCP Mode	298
Using Buffers in UDP Mode	299
TX and RX Buffer Memory Status	300
Receiving Data in TCP Mode	301
Receiving Data in UDP Mode	303
Sending TCP and UDP Data	304
"Split Packet" Mode of TCP Data Processing	306
Handling Buffer Overruns	307
Redirecting Buffers	308
Sinking Data	309
Working With Inband Commands	309
Inband Message Format	309
Inband-related Buffers (CMD, RPL, and TX2)	310
Processing Inband Commands	311
Sending Inband Replies	313
Using HTTP	314
HTTP-related Buffers	315
Setting the Socket for HTTP	317
Socket Behavior in the HTTP Mode	318
Including BASIC Code in HTTP Files	319
Generating Dynamic HTML Pages	320
URL Substitution	322

Working with HTTP Variables.....	323
Simple Case (Small Amout of Variable Data).....	323
Complex Case (Large Amount of Variable Data).....	324
Details on Variable Data	326
Properties, Methods, and Events.....	327
.Acceptbcast Property	327
.Allowedinterfaces Property.....	327
.Bcast R/O Property	328
.Close Method	328
.Cmdbuffrq Method	329
.Cmdlen R/O Property	329
.Connect Method	330
.Connectiontout Property	330
.Currentinterface R/O Property.....	331
.Discard Method	331
.Endchar Property	331
.Escchar Property	332
.Event R/O Property (Obsolete).....	332
.Eventsimple R/O Property (Obsolete).....	332
.Getdata Method	333
.Gethttprqstring Method	333
.Getinband Method	334
.Httpmode Property	334
.Httpnoclose Property	335
.Httpportlist Property	335
.Httprqstring R/O Property.....	336
.Inbandcommands Property.....	337
.Inconenabledmaster Property.....	337
.Inconmode Property	338
.Localport R/O Property	338
.Localportlist Property	339
.Newtxlen R/O Property	339
.Nextpacket Method	339
.Notifysent Method	340
.Num Property	340
.Numofsock R/O Property.....	341
.Outport Property	341
.On_sock_data_arrival Event.....	342
.On_sock_data_sent Event.....	342
.On_sock_event Event	343
.On_sock_inband Event	343
.On_sock_overrun Event	343
.On_sock_postdata	344
.On_sock_tcp_packet_arrival Event.....	344
.Protocol Property	345
.Reconmode Property	345
.Redir Method	346
.Remoteip R/O Property	347
.Remotemac R/O Property.....	348
.Remoteport R/O Property.....	348
.Reset Method	348
.Rplbuffrq Method	349
.Rplfree R/O Property	350
.Rplen R/O Property	350
.Rxbuffrq Method	350
.Rxbuffsize R/O Property	351
.Rxclear Method	351
.Rxpacketlen R/O Property.....	352
.Rxlen R/O Property	352

.Send Method	353
.Setdata Method	353
.Setsendinband Method	354
Sinkdata Property	354
.Splittcpackets Property	355
.State R/O Property	355
.Statesimple R/O Property	358
.Targetbcast Property	359
.Targetinterface Property	359
.Targetip Property	359
.Targetport Property	360
.Toutcounter R/O property	360
.Tx2buffrq Method	361
.Tx2len R/O Property	362
.Txbuffrq Method	362
.Txbuffsize R/O Property	363
.Txclear Method	363
.Txfree R/O Property	363
.Txlen R/O Property	364
.Urlsubstitutes	364
.Varbuffrq Method	365
IO Object	365
Overview	365
Line/Port Manipulation With Pre-selection	366
Line/Port Manipulation Without Pre-selection	367
Controlling Output Buffers	368
Working With Interrupts	369
Properties, Events, Methods	370
.Enabled Property (Selected Platforms Only)	370
.Intenabled Property	370
.Intnum Property	371
.Invert Method	371
.Lineget Method	371
.Lineset Method	372
.Num Property	372
On_io_int Event	373
.Portenabled Property (Selected Platforms Only)	373
.Portget Method	373
.Portnum Property	374
.Portset Method	374
.Portstate property	375
.State Property	375
Romfile Object	375
.Find Method	377
.Getdata Method	378
.Offset R/O Property	378
.Open Method	379
.Pointer Property	379
.Size R/O Property	379
Stor Object	380
.Base Property	381
.Getdata Method (previously .Get)	381
.Setdata Method (previously .Set)	382
.Size R/O Property	383
Pat Object	384
.Channel Property	385
.Greenmap Property	385
.On_pat Event	386
.Play Method	386

.Redmap Property	387
Beep Object	387
.Divider Property	388
.On_beep Event	388
.Play Method	389
RTC Object	389
.Getdata Method (Previously .Get)	390
.Running R/O Property	391
.Setdata Method (Previously .Set)	391
LCD Object	392
Overview	393
Understanging Controller Properties	393
Preparing the Display for Operation	395
Working With Pixels and Colors	395
Lines, Rectangles, and Fills	396
Working With Text	397
Raster Font File Format	400
Displaying Images	404
Improving Graphical Performance	405
Supported Controllers/Panels	408
Samsung S6B0108 (Winstar WG12864F)	408
Solomon SSD1329 (Ritdisplay RGS13128096)	410
Himax HX8309 (Ampire AM176220)	411
Properties and Methods	412
.Backcolor Property	414
.Bitsperpixel R/O Property	414
.Bluebits R/O Property	415
.Bmp Method	415
.Enabled Property	416
.Error R/O Property	417
.Fill Method	417
.Filledrectangle Method	418
.Fontheight R/O Property	418
.Fontpixelpacking R/O Property	419
.Forecolor Property	419
.Getprintwidth Method	420
.Greenbits R/O Property	420
.Height Property	421
.Horline Method	421
.Inverted Property	422
.Iomapping Property	422
.Line Method	422
.Linewidth Property	423
.Lock Method	423
.Lockcount R/O Property	424
.Paneltype R/O Property	424
.Pixelpacking R/O Property	425
.Print Method	425
.Printaligned Method	426
.Rectangle Method	427
.Redbits R/O Property	427
.Rotated Property	428
.Setfont Method	428
.Setpixel Method	429
.Textalignment Property	429
.Texthorizontalspacing Property	430
.Textorientation Property	430
.Textverticalspacing Property	431
.Unlock Method	431

.Verline Method	432
.Width Property	432
Fd Object	433
Overview	433
Sharing Flash Between Your Application and Data.....	434
Fd. Object's Status Codes.....	435
File-based Access	436
Formatting the Flash Disk.....	436
Disk Area Allocation Details.....	437
Mounting the Flash Disk	439
File Names and Attributes.....	440
Checking Disk Vitals	441
Creating, Deleting, and Renaming Files.....	441
Reading and Writing File Attributes.....	442
Walking Through File Directory.....	442
Opening Files	443
Writing To and Reading From Files.....	444
Removing Data From Files.....	445
Searching Files	446
Closing Files	448
Direct Sector Access	448
Using Checksums	450
Upgrading the Firmware/Application.....	452
File-based and Direct Sector Access Coexistence.....	453
Prolonging Flash Memory Life	453
Ensuring Disk Data Integrity	454
Properties and Methods	456
.Availableflashspace R/O Property.....	457
.Buffernum Property	458
.Capacity R/O Property	458
.Checksum Method	459
.Close Method	460
.Copyfirmware Method	460
.Cutfromtop Method	461
.Create Method	462
.Delete Method	463
.Filenum Property	463
.Fileopened R/O Property.....	464
.Filesize R/O Property	464
.Find Method	464
.Flush Method	466
.Format Method	467
.Getattributes Method	467
.Getbuffer Method	468
.GetData Method	469
.Getfreespace Method	470
.Getnextdirmember Method	470
.Getnumfiles Method	471
.Getsector Method	472
.Laststatus R/O Property	472
.Maxopenedfiles R/O Property.....	473
.Maxstoredfiles R/O Property.....	474
.Mount Method	474
.Numservicesectors R/O Property.....	475
.Open Method	475
.Pointer R/O Property	476
.Ready R/O Property	476
.Rename Method	477
.Resetdirpointer Method	478

.Sector R/O Property	478
.Setattributes Method	478
.Setbuffer Method	479
.Setdata Method	480
.Setfilesize Method	481
.Setpointer Method	482
.Setsector Method	483
.Totalsize R/O Property	483
Kp Object	484
Possible Keypad Configurations.....	484
Key States and Transitions.....	486
Preparing the Keypad for Operation.....	487
Servicing Keypad Events	489
Properties, Methods, Events.....	492
.Autodisablecodes Property.....	492
.Enabled Property	492
.Longpressdelay Property.....	493
.Longreleasedelay Property.....	493
.On_kp Event	494
.On_kp_overflow Event	494
.Pressdelay Property	495
.Releasedelay Property	495
.Repeatdelay Property	496
.Returnlinesmapping Property.....	496
.Scanlinesmapping Property.....	497
WIn Object	497
Migrating From the WA1000.....	498
Overview	499
Wi-Fi Parlance Primer	500
WIn Tasks	500
WIn State Transitions	503
Brining Up Wi-Fi Interface.....	504
Allocating Buffer Memory	505
Applying Reset	506
Configuring Interface Lines	507
Setting MAC Address (Optional).....	507
Selecting Domain	508
Booting Up the Hardware	508
Setting IP, Gateway, and Netmask (Optional).....	509
Setting TX Power (Optional).....	509
Scanning for Wi-Fi Networks	509
Setting WEP Mode and Key.....	510
Associating With Selected Network.....	511
Creating Own Ad-hoc Network.....	512
Communicating via WIn Interface.....	512
Disassociating From the Network	512
Terminating Own Ad-hoc Network	512
Rebooting	513
Detecting Disassociation or Offline State.....	513
Properties, Methods, Events.....	513
.Associate Method	513
.Associationstate R/O Property	514
.Boot Method	515
.Bssmode Property	515
.Buffrq Method	516
.Buffsize R/O Property	516
.Clkmap Property	517
.Csmap Property	517
.Defaultibsschannel Property.....	517

.Dimap Property	517
.Disassociate Method	518
.Domain Property	518
.Domap Property	519
.Enabled R/O Property	519
.Gatewayip Property	520
.Ip Property	520
.Mac Property	520
.Netmask Property	521
.Networkstart Method	521
.Networkstop Method	522
On_wln_event Event	522
On_wln_task_complete Event	523
.Rssi R/O Property	524
.Scan Method	524
.Scanresultbssid R/O Property	525
.Scanresultbssmode R/O Property	525
.Scanresultchannel R/O Property	526
.Scanresultrssi R/O Property	526
.Scanresultssid R/O Property	526
.Settxpower Method	527
.Setwep Method	527
.Ssid Property	528
.Task R/O Property	528
.Wepkey1 Property	529
.Wepkey2 Property	529
.Wepkey3 Property	529
.Wepkey4 Property	529
.Wemode Property	530

Update History (for this Manual)**530**

Taiko R2

Last update: 29JUL2009

[Legal Information](#) 

[Manual Update History](#) 

Taiko is a solution which allows you to create programs for Tibbo modules capable of running TiOS (Tibbo Operating System), and products based on these modules.

With Taiko, you write your program in a language called Tibbo Basic (a close relative of any other BASIC you might already know), using a PC software called TIDE - Tibbo Integrated Development Environment. Your program is then compiled into a binary file and uploaded onto a Tibbo module. The Virtual Machine of TiOS then executes this binary.

Taiko allows you to easily create programs for a variety of Tibbo-based products. These may include:

- Alarm Panels
- Security Systems (Access control terminals, etc)
- Data Collection terminals, such as time clocks
- Sensor monitors
- Interface converters
- Vending machines
- Industrial process controllers

The solutions created with Taiko are very flexible. They are written using a language similar to BASIC, and are stored on a Tibbo module separately from the core OS of the module (TiOS). This allows for simple modification of your device functionality, even by the end-user (if you so allow).

Tibbo Basic itself is exactly the same for all TiOS-enabled devices. Hardware differences are expressed through so-called *platforms*. Change the platform, and you're programming for a different device.

Documentation Map

The documentation for Taiko includes:

[Overview](#)  - The theory and background behind Taiko.

[Getting Started](#)  - An example starter project.

[Programming with TIDE](#)  -- An overview of TIDE itself, debug facilities, etc.

[Language Reference](#)  -- Systematically covers Tibbo Basic statements, keywords and operators.

[Development Environment](#)  -- Systematically covers TIDE GUI elements.

[Glossary of Terms](#)  -- Contains some basic terms used in Taiko.

[Platforms](#)  -- Platform-specific documentation for each target device.

Legal Information

Tibbo Technology ("TIBBO") is a Taiwan corporation that designs and/or manufactures a number of hardware products, software products, and applications ("PRODUCTS"). In many cases, Tibbo PRODUCTS are combined with each other

and/or third-party products thus creating a PRODUCT COMBINATION.

Whereas you (your Company) wish to purchase any PRODUCT from TIBBO, and/or whereas you (your Company) wish to make use of any documentation or technical information published by TIBBO, and/or make use of any source code published by TIBBO, and/or consult TIBBO and receive technical support from TIBBO or any of its employees acting in an official or unofficial capacity,

You must acknowledge and accept the following disclaimers:

1. Tibbo does not have any branch office, affiliated company, or any other form of presence in any other jurisdiction. TIBBO customers, partners and distributors in Taiwan and other countries are independent commercial entities and TIBBO does not indemnify such customers, partners or distributors in any legal proceedings related to, nor accepts any liability for damages resulting from the creation, manufacture, importation, advertisement, resale, or use of any of its PRODUCT or PRODUCT COMBINATION.
2. BASIC-programmable devices ("PROGRAMMABLE DEVICES") manufactured by TIBBO can run a variety of applications written in Tibbo BASIC ("BASIC APPLICATIONS"). Combining a particular PROGRAMMABLE DEVICE with a specific BASIC APPLICATION, either written by TIBBO or any third party, may potentially create a combinatorial end product ("END PRODUCT") that violates local rules, regulations, and/or infringes an existing patent granted in a country where such combination has occurred or where the resulting END PRODUCT is manufactured, exported, advertised, or sold. TIBBO is not capable of monitoring any activities by its customers, partners or distributors aimed at creating any END PRODUCT, does not provide advice on potential legal issues arising from creating such END PRODUCT, nor explicitly recommends the use of any of its PROGRAMMABLE DEVICES in combination with any BASIC APPLICATION, either written by TIBBO or any third party.
3. TIBBO publishes a number of BASIC APPLICATIONS and segments thereof ("CODE SNIPPETS"). The BASIC APPLICATIONS and CODE SNIPPETS are provided "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of BASIC APPLICATIONS and CODE SNIPPETS resides with you. BASIC APPLICATIONS AND CODE SNIPPETS may be used only as a part of a commercial device based on TIBBO hardware. Modified code does not have to be released into the public domain, and does not have to carry a credit for TIBBO. BASIC APPLICATIONS and CODE SNIPPETS are provided solely as coding aids and should not be construed as any indication of the predominant, representative, legal, or best mode of use for any PROGRAMMABLE DEVICE.
4. BASIC-programmable modules ("PROGRAMMABLE MODULES"), such as the EM1000 device, are shipped from TIBBO in either a blank state (without any BASIC APPLICATION loaded), or with a simple test BASIC APPLICATION aimed at verifying correct operation of PROGRAMMABLE MODULE's hardware. All other BASIC-programmable products including boards, external controllers, and developments systems ("NON-MODULE PRODUCTS"), such

as the DS1000 and NB1000, are normally shipped with a BASIC APPLICATION pre-loaded. This is done solely for the convenience of testing by the customer and the nature and function of pre-loaded BASIC APPLICATION shall not be construed as any indication of the predominant, representative, or best mode of use for any such NON-MODULE PRODUCT.

5. All specifications, technical information, and any other data published by TIBBO are subject to change without prior notice. TIBBO assumes no responsibility for any errors and does not make any commitment to update any published information.
6. Any technical advice provided by TIBBO or its personnel is offered on a purely technical basis, does not take into account any potential legal issues arising from the use of such advice, and should not be construed as a suggestion or indication of the possible, predominant, representative, or best mode of use for any Tibbo PRODUCT.
7. Neither TIBBO nor its employees shall be held responsible for any damages resulting from the creation, manufacture, or use of any third-party product or system, even if this product or system was inspired, fully or in part, by the advice provided by Tibbo staff (in an official capacity or otherwise) or content published by TIBBO or any other third party.
8. TIBBO reserves the right to halt the production or availability of any of its PRODUCTS at any time and without prior notice. The availability of a particular PRODUCT in the past is not an indication of the future availability of this PRODUCT. The sale of the PRODUCT to you is solely at TIBBO's discretion and any such sale can be declined without explanation.
9. TIBBO makes no warranty for the use of its PRODUCTS, other than that expressly contained in the Standard Warranty located on the Company's website. Your use of TIBBO PRODUCTS is at your sole risk. TIBBO PRODUCTS are provided on an "as is" and "as available" basis. TIBBO expressly disclaims the warranties of merchantability, future availability, fitness for a particular purpose and non-infringement. No advice or information, whether oral or written, obtained by you from TIBBO shall create any warranty not expressly stated in the Standard Warranty.
10. LIMITATION OF LIABILITY. BY USING TIBBO PRODUCTS YOU EXPRESSLY AGREE THAT TIBBO SHALL NOT BE LIABLE TO YOU FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES, INCLUDING, BUT NOT LIMITED TO, DAMAGES FOR LOSS OF PROFITS, GOODWILL, OR OTHER INTANGIBLE LOSSES (EVEN IF TIBBO HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES) RESULTING FROM THE USE OR THE INABILITY TO USE OF TIBBO PRODUCTS.
11. "Tibbo" is a registered trademark of Tibbo Technology, Inc.
12. Terms and product names mentioned on TIBBO website or in TIBBO documentation may be trademarks of others.

Overview

Below is a summary of the major fundamentals and theory behind TIDE. This may sound intimidating, but it's actually quite simple. You *should* at least skim over the material herein, because it explains much of what comes next. In here you will find:

- [Our Language Philosophy](#)⁴
- [System Components](#)⁷
- [Objects](#)⁸ (a very brief overview)
- [Events](#)⁸

Our Language Philosophy

Several principles have guided us through the development process of Tibbo Basic. Understanding them would help you understand this manual better, and also the language itself. See below:

A Bit of History

Years ago, programming for the PC was the nearly exclusive domain of engineers. The languages traditionally available, such as C, simply required you to be an engineer to program.

However, one day something interesting happened. Visual Basic* and Delphi** saw the light of day. And that changed quite a lot on the PC front. Suddenly, people who were not engineers were finding out that they could actually create something cool on their PC. You could say VB* and Delphi democratized the PC software market.

The situation on the embedded systems market today is quite similar to the situation which existed for the PC market in the pre-VB era. Many embedded systems vendors do offer customizable or programmable solutions -- but to implement those solutions, you would really have to be an engineer and know C/C ++ quite well. So, there was clearly a need for an easy-to-use programming system which would democratize this market, as well.

Principle One: Easy To Write, Easy to Debug

Choosing BASIC as our inspiration was the natural thing to do, for us. It's a language which doesn't require you to be a professional engineer. It is easy to understand. This is why it is embedded into many non-programmer products, such as the Office suite. So we went for BASIC.

Another part of the user experience, and a major one, too, is debugging. Writing your application is just half the job. You also need to debug it and for embedded systems, this is where things typically start getting rough around the edges. Many times you have to buy expensive tools, such as ICE machines (In-Circuit Emulators), just to figure out what your code is doing. Sometimes you don't even have the luxury of such a machine, and you actually debug by guessing and trying different things in your code.

With our system, one of our major goals was to offer a user experience which is close to debugging on the PC -- without the need for special tools, such as an ICE machine.

While your program is running on the target (embedded device), you actually see how it runs *on your PC*. You can step through it, jump to specific functions, check values of variables etc -- all from the comfort of your own PC.

Principle Two: Easy Doesn't Mean Sloppy

Some modern programming languages use certain techniques to make life 'easier' for programmers. They might not require the programmer to explicitly declare the variables he's going to use ('implicit declaration'), or might do away with the need to specify the type for the variable (i.e, use 'variant variables' which can contain anything).

This has several disadvantages. For one, it is just sloppy. After several days of writing code like that, a programmer might not have a very clear-cut idea of what his program is doing, or where things come from. While this is something which may be subject to debate, the next disadvantage is quite real:

This is simply wasteful programming. These techniques can consume quite a lot of resources, specifically memory. On the PC, a variant used to store just 2 bytes of data might take up to 100 bytes. This isn't a problem, because PCs have so much memory these days that it is barely felt.

However, embedded systems are often low-cost and bare-bones, so physical memory is a truly valuable resource. Waste too much of it -- and you would find that your code can do very little. But manage it prudently, and your code will be capable of quite impressive feats even on your 'low-power' embedded system.

So our systems requires you to be more organized. The effort is worth it.

Principle Three: The Purity of Language

Programming systems on the PC usually make no clear distinction between the 'pure' language constructs which perform calculations and control program flow, and hardware-dependant input/output. For example, many languages contain a print statement which prints something to the screen.

Since all PCs in the world are similar, this works. However, this makes little sense for embedded platform, which have vastly different input/output resources.

Depending on the device, it may or may not have a screen, a serial port, networking etc etc.

In our system, we separated the language itself (what we call *the core language*) from the input/output of a particular device. Thus, the language itself remains the same, no matter what device you are programming for. The input/output part is hardware dependant, and changes from platform to platform.

When writing for a specific platform, you are provided with a set of platform-specific *objects*. These provide rich functionality and allow you to do 'real' work, such as printing messages to the serial port, communicating on the Internet or controlling motors and sensors.

Ideally, Tibbo Basic could run on a fridge just as well as it could run on a time and attendance terminal.

Principle Four: Thin and Agile

A lot of embedded systems are built by scaling down larger desktop systems, and it shows. What's the point of using a super-fast processor if you load it with dozens of layers of nested calls?

All the code TiOS includes has been designed from scratch for running on a very simple processor, and optimized for control applications. It has been crafted to have the minimum possible ROM and RAM footprint and to run as fast as possible.

We built TiOS with Pareto's principle in mind. In other words, if a certain functionality is required by only 5% of applications and yet its existence adds 90% overhead, we did not include it. For example, our BASIC only supports integer

calculations. Most other BASIC versions, by default, operate using floating point arithmetic, but these are not usually useful to embedded control (and even get in the way). Another decision was to use a static memory model for procedure variables. Memory is not allocated and deallocated dynamically -- It is assigned on compile-time, which results in great performance improvements.

Principle Five: No B.S

... that is, no babysitting. Development systems intended for rapid application development on the PC will often try to handle every little error or problem the programmer may encounter. If a variable overflows, for example, they will *halt* execution and pop up an error to let him know. This makes sense for a PC-based product, because you are right there to see it halt.

However, when you are creating an embedded system, you expect it to run *at all times*, without halting. Nobody will be there to see any errors and babysit your system. Your device is simply expected *to work*.

This is a major difference also for the development process. In essence, since the whole language is built this way, you will also get much less errors even when doing seemingly 'strange' things, such as putting large values into variables that cannot hold them. The language will deal with it silently, in a very predictable and logical way -- but will not pop up an error.

Principle Six: Event-Driven Programming

Users of VB and Delphi and other Windows-based tools will find this principle familiar. However, if most of your experience with BASIC was under DOS, you might find this slightly odd. Under DOS, you would expect a program to begin from the beginning, then continue and stop. They execute from top to bottom. This may be called *linear execution*.

For Tibbo Basic, this is not the case. The programs you will write will be *event-driven*. Your program will consist of a number of *event handlers* which will be fired (invoked) in response to specific things which happen to your system in real life. If your platform was a fridge, you might want to write a handler for a 'door opening' event. When the door is opened, an event is generated, and an event handler, with your code in it, is fired.

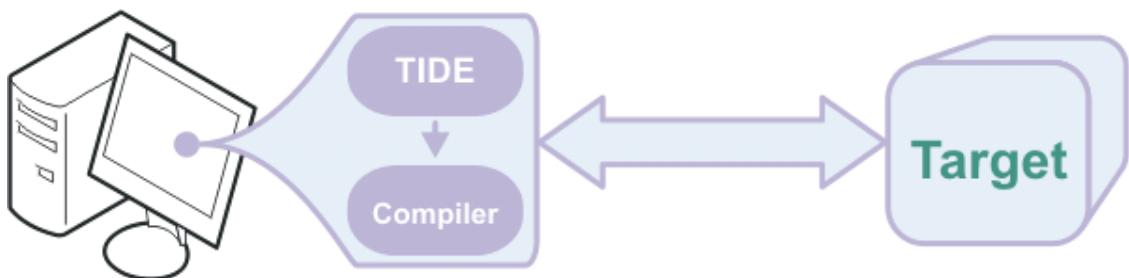
So, you could say that your event-driven application has no beginning and no end. Event handlers are called when events are generated, and in the order in which they were generated.

* Windows, Visual Basic and VB are registered trademarks of Microsoft Corporation Inc.

** Delphi is a registered trademark of Borland Inc.

System Components

Taiko is a compound system. It consists of the following components:



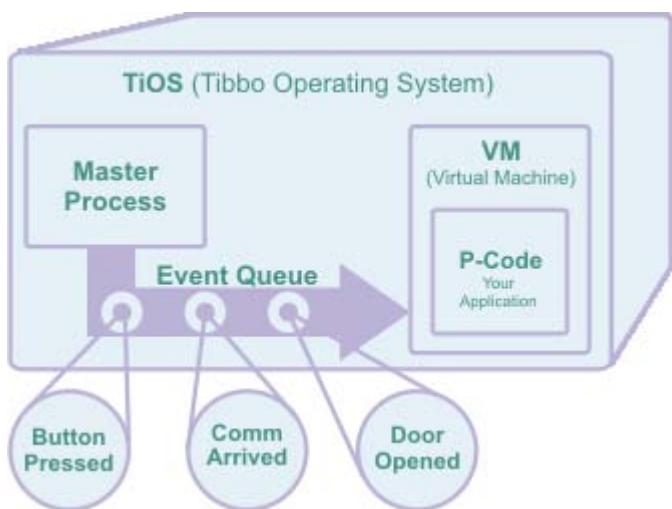
TIDE is an acronym for Tibbo Integrated Development Environment. This is the PC program in which you will write your applications and compile them, and from which you will upload them to your target and debug them.

The **compiler** is a utility program, used by TIDE. The compiler processes your project files and creates an *executable binary file* (with a .tpc suffix, for Tibbo PCode).

The **target** is a separate hardware device, on which your program actually runs. When debugging code, it is connected to your computer running TIDE (see the link above) and TIDE can monitor and control it. This is called *cross-debugging*.

As covered under [Our Language Philosophy](#)^[4], Tibbo Basic is capable of running on various hardware devices. Each type of hardware device on which Tibbo Basic runs is called a *platform*.

And now, the anatomy of the target:



The **target** runs an operating system called TiOS (Tibbo Operating System).

TiOS runs two processes. One is the **Master Process**. This is the process which is in charge of communications (including communications with TIDE) and of generating events. The second process, which is under the control of the Master Process, is called the **VM** (Virtual Machine).

The VM is what actually executes your application. In essence, the VM is a processor implemented in firmware, which executes the compiled form of your application. The instructions it understands are called **P-Code**, which is short for pseudo-code. This is what the compiler produces. It is called pseudo-code because

it is not native binary code which the hardware processor can understand directly; instead, it is interpreted by the VM.

Since the VM is under the complete control of the Master Process, the actual hardware processor will not crash because of an error in your Tibbo Basic application. Your application may operate incorrectly, but you still will be able to debug it. The Master Process can stop or restart the Virtual Machine at will, and can exchange debug information with TIDE, such as report current execution state, variable values, etc.

Simply put, you can think of the VM as a sort of a 'sandbox' within the processor. Your application can play freely, without the possibility of crashing or stalling TiOS due to some error.

The **queue** is used to 'feed' your program with [events](#)⁸ which it should handle. The Master Process monitors the various interfaces of the platform and generates events, putting them into the queue. The Virtual Machine extracts these events from the other side of the queue and feeds your program with them. Various parts of your program execute in response to events.

Objects

Objects represent the various component part of your platform. For example, a platform with a serial port might have a *ser* object. A platform can be described as a collection of objects.

Under Tibbo Basic, the set of object you get for each platform is fixed. You cannot add new objects or create multiple instances of the same object.

Objects have properties, methods and events. A *property* can be likened to an attribute of the object, and a *method* is an action that the object can perform. [Events](#)⁸ are described in the next section.

Objects are covered in further detail under [Objects, Events and Platform Functions](#)⁷⁸.

Events

An *event* is something which happens to an object. Plain and simple. A fridge might have door object with an *on_door_open* event, and a paper shredder might have a detector object with an *on_paper_detected* event.

Events are a core concept in Tibbo Basic. They are the primary way in which code gets executed.

The target device maintains an *event queue*. All events registered by the system go into this queue. On the other end of the queue, the Virtual Machine takes out one event at a time and calls an event handler for each event.

Event handlers are subroutines in your code which are 'fired' (executed) to handle an event. Often, event handlers contain function calls which run other parts of the program.

While processing an event, other events may happen. These events are then queued for processing, and patiently wait for the first event to complete before beginning execution.

All Tibbo Basic programs are single-threaded, so there is only one event queue. All events are executed in the exact order in which they were queued.

It may sometimes seem that some events should get priority over other events. This functionality is not supported under Tibbo Basic. This is not crucial, as events

tend to execute very quickly, and the queue ensures events are not forgotten.

Getting Started

Below is a walk-through for a starter project which is written specifically for the EM202-EV and DS202.

Once you are done with this project, you will be able to press the button on the EM202-EV or DS202 and watch the LEDs blink "Hello World!" in Morse code.



This project would actually run also on the EM202, EM200 and EM120 modules. However, these modules cannot work on their own, and you cannot easily test with them.

Preparing Your Hardware

Preparing a DS202

Before starting to use TIDE, you should upload the correct firmware to a DS202. Perform the following steps:

- Get tios_EM202_xxx.bin firmware file (the latest version) from the Tibbo website. _100 in this filename stands for version 1.00, for example.
- Connect the DS202 to power (preferably, use adaptor supplied by Tibbo).
- Connect the DS using a network cable (WAS-1499 or similar) to the same hub your computer is connected to, or directly to the computer with a cross network cable (WAS-1498 or similar).
- Make sure your local firewall (such as the XP SP2 firewall) is disabled or does not block broadcast UDP messages. This is essential for communications between TIDE and the DS202 while debugging.
- Run Device Explorer (Start > Programs > Tibbo > Tibbo IDE > Device Explorer).
- You should see your device on the list. Select it.
- Click Upload > Load Firmware Through the Network.
- Select the firmware file, and click OK.
- The firmware will now be uploaded.
- For some firmware versions, you now have to manually reboot the DS (Disconnect and reconnect the power cable). The red Status LED should now blink rapidly. This is OK -- it means the TiOS firmware is loaded and the application program memory is empty.
- Proceed to [Starting a New Project](#).



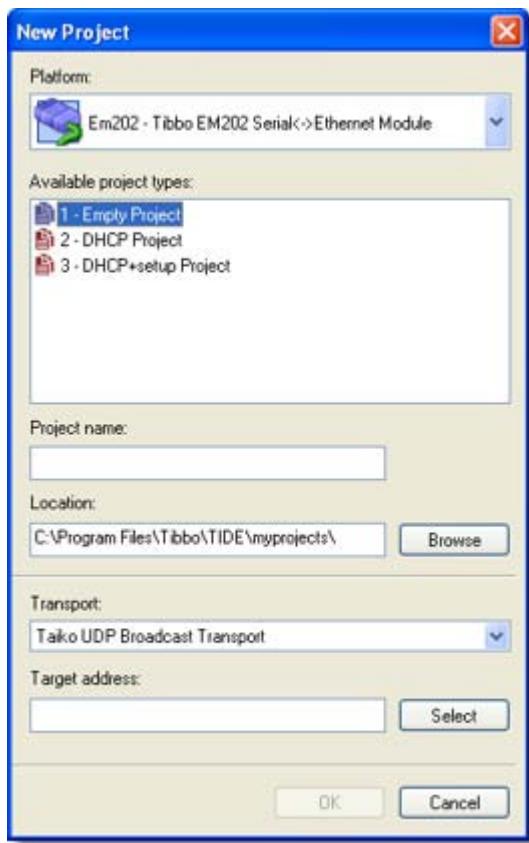
Of course, once you upload a device with the TiOS firmware, it is no longer a Device Server! So you cannot see it under DS Manager. You could program it so it would respond to DS Manager -- but by default it is a 'clean slate', and does not respond to DS Manager broadcasts.



If for some reason you cannot perform a network upload, you can perform a serial upload by selecting Upload > Load firmware Through the Serial Port. You will then be prompted to select a COM port, turn the device off and turn it back on while pressing the SETUP button. Upload will then commence.

Starting a New Project

To begin a new project, select File > New. You will be presented with the following dialog:



Platform: Select EM202 (you can use EM1000 as well)

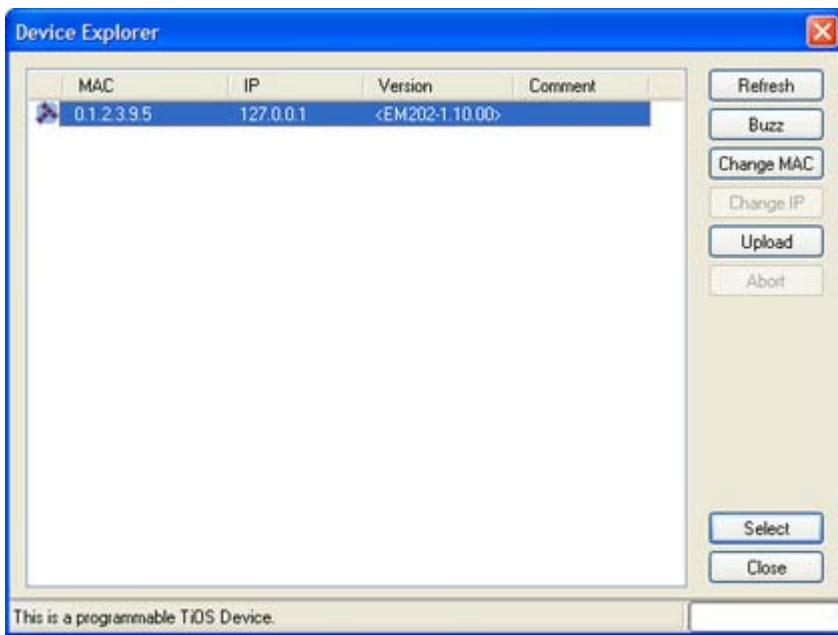
Available project types: Select Empty Project.

Project name: Type 'Hello World'.

Location: Leave untouched, unless you have a good reason to change it.

Transport: leave it as is ("Taiko UDP Broadcast Transport")

Target Address: Click Browse. You will be presented with the following dialog:



The number (hopefully) displayed is the MAC address of your target. If you select it and click Buzz, you should see the LED pattern on your target switch off momentarily. This means it is correctly detected.



If you see nothing in this dialog, it means your target isn't in communication with the computer. This is probably a power problem, or a networking problem. Perhaps you have a local firewall on the computer which blocks UDP broadcasts, such as the Windows XP Firewall. To fix this, disable the firewall or configure it to open a specific port.

Once you have located your target, click Select. You will be returned to the previous dialog, and the MAC address for your target will appear under Target Address.

You have now specified all of the required settings for a new project. Click OK and proceed.

Writing Code

Once you have started your new project, you will be presented with a blank file (main.tbs).

We will now begin writing the actual code in this file. We will construct this project from beginning to end, step by step. For your convenience, the end of this section contains a complete copy of the project without comments. You can copy and paste the whole thing into TIDE, or just copy and paste the commented sections one by one as they appear below.

Here goes:

```

' Comments cannot spill over to the next line. If you see this happening
in this manual, it is a result of the help system -- not an actual
feature.

Dim hello_world As String ' define a variable which will hold the whole
pattern we will play.

Dim length, play_position As Integer ' length is a calculated integer
which will contain the whole length of the string we will play, and
play_position will contain our current position in this string (how much
we have played so far).

Const PAT_PLAY_CHUNK_LENGTH = 15 ' define a constant for the size of the
chunk we will play. We will play one chunk of the pattern at a time, and
then move on to the next chunk. Each chunk is 15 'steps' long.

Declare Sub play_Next ' let the compiler know that there is a sub called
play_next. This sub will be used in code before being created so we must
declare it.

```

Notice that we are defining a *chunk* above. The reason for this is that we are going to play quite a long and complex pattern (over 130 steps in length), but the pattern object (pat.) used to play the pattern only supports patterns of up to 16 steps. So we have to play our pattern in parts, one after the other, and track our progress through the pattern (this is what the counters are for).

So far, we have prepared the ground. Let us move to the first piece of executable code:

```

sub on_sys_init ' event handler for the init event. Fires whenever the
device powers on.
    hello_world = ' here we define the contents of our string, in morse.

        'R is Red LED, G is Green LED. GGG means a long pulse of the
green LED (line). R means a short pulse of the Red LED (dot). Line (-)
means both off.

        'HELLO .... . .-. .-.. ---
        "R-R-R-R---R---R-GGG-R-R---R-GGG-R-R---GGG-GGG-GGG" +
        "-----" + ' A period of silence between words
        'WORLD .-- --- .-. .-.. -..
        "R-GGG-GGG---GGG-GGG-GGG---R-GGG-R---R-GGG-R-R---GGG-R-R" +
        "-----" +
        '! ..-..
        "R-R-GGG-GGG-R-R-"
length = len(hello_world) ' Calculate total length of string.
play_position = 1 ' Initialize play_position as we haven't played
anything yet.
end sub

```

We will now write the event handlers for our code.

First, we want the pattern to start playing whenever you press the button. For this, our platform offers a button object, which generates an `on_button_pressed` event. Instead of typing, you can create the event handler for this event by double-clicking on the event name in the [project tree](#)^[130].

```
sub on_button_pressed ' event handler fired whenever the button is pressed
    play_position = 1 ' start playing from the beginning of the pattern
    play_next ' call the routine which plays the next chunk (the first
    chunk, in this case)
end sub
```

Notice that the play_next routine is not yet defined. In our code, it is first used and then defined. This is why we have to [declare](#)⁷⁹ it at the beginning.

Now, let us move on to the next event handler:

```
sub on_pat ' this fires whenever a pattern (a chunk, in our case)
finishes playing.
    play_next ' call the routine which plays the next chunk
end sub
```

We have now completed writing our event handlers. Our program now knows what it's supposed to do whenever you press the button, and whenever a chunk of the pattern finishes playing. It just doesn't know *how* to do it yet. This comes next:

```
sub play_next ' plays the next chunk of our large pattern.
    if length < play_position then exit sub ' if we have reached the end
    of the pattern, stop.

    dim chunk_len as integer ' internal integer for the length of current
    chunk to be played.

    chunk_len = length - play_position + 1 ' calculate how much of the
    large string is left.

    if chunk_len > PAT_PLAY_CHUNK_LENGTH then chunk_len =
PAT_PLAY_CHUNK_LENGTH ' if too much is left, we bite off only a chunk we
    can process.

    dim chunk as string ' will contain the chunk which will actually play
    now.
    chunk = mid(hello_world, play_position, chunk_len) ' chunk is the
    part of hello_world which begins at play_position and is as long as
    chunk_len.

    pat.play(chunk, YES) ' Play this chunk. YES means the pattern may be
    interrupted -- you can press the button while the pattern is playing, and
    it will start again from the top.

    play_position = play_position + chunk_len ' advance play_position to
    account for the chunk we played.

end sub
```

Here is the whole project, without comments:

```

'=====
'
'           HELLO WORLD IN MORSE CODE (for EM202-EV, DS202)
'=====

dim hello_world as string
dim length, play_position as integer

const PAT_PLAY_CHUNK_LENGTH = 15

declare sub play_next

'

-----
sub on_sys_init
    hello_world =
        "R-R-R-R---R---R-GGG-R-R---R-GGG-R-R---GGG-GGG-GGG" +
        "-----" +
        "R-GGG-GGG---GGG-GGG-GGG---R-GGG-R---R-GGG-R-R---GGG-R-R" +
        "-----" +
        "R-R-GGG-GGG-R-R-"
    length = len(hello_world)
    play_position = 0
end sub

'

-----
sub on_button_pressed
    play_position = 1
    play_next
end sub

'

-----
sub on_pat
    play_next
end sub

'

-----
sub play_next
    if length < play_position then exit sub

    dim chunk_len as integer
    chunk_len = length - play_position + 1
    if chunk_len > PAT_PLAY_CHUNK_LENGTH then chunk_len =
PAT_PLAY_CHUNK_LENGTH

    dim chunk as string
    chunk = mid(hello_world, play_position, chunk_len)
    pat.play(chunk, YES)
    play_position = play_position + chunk_len
end sub

```

Building, Uploading and Running

Once you are done with writing your project, it is time to build, upload and run it. These three operations can be done by pressing F5.

Press F5 and wait. You will see your project compiling. The [output pane](#)^[129] will

display any errors (if you copied the project as it is, there should be no errors).

The [status bar](#)^[26] will show you the project building, uploading, and running.

Once the status bar says RUNNING, you may press the button on your device to see it blink "Hello World" in Morse.

For further information about these topics, please see [Making, Uploading and Running an Executable Binary](#)^[26] and [Debugging Your Project](#)^[28] below.

Compiling a Final Binary

The binary executable file you compiled in the previous step is called a [debug binary](#)^[27]. This type of binary is used while creating your project and debugging it.

When you decide your project is ready to be deployed in the real world, you should compile a [release binary](#)^[27]. To do this, select Project > Settings and uncheck the Debug version checkbox.

The next time you will press F5, a release binary will be created and uploaded to your target. It will automatically start running and will not provide any debug information.

This release binary file also remains on your hard drive, inside your project folder (see [Starting a New Project](#)^[10]). You may take it and upload it to any number of DSees.

Programming with TIDE

The topics below attempt to give you a general understanding about working with Tibbo Basic. An attempt has been made to lay them out as logically as possible; it would be advised to just read them from top to bottom and follow the links every time you don't understand a term.

The section called [Managing Projects](#)^[15] provides an overview of the general structure of a Tibbo Basic project, and also discusses the debugging process.

The next section, [Programming Fundamentals](#)^[39], then delves into the specifics of Tibbo Basic programming, including the differences between Tibbo Basic and other languages you may know.

Managing Projects

Each program you will make with Tibbo Basic is actually a project. Projects include certain files, and have a specific structure. They are compiled into binary files, uploaded onto your target and debugged. Below you will learn about:

- [Creating, Opening and Saving Projects](#)^[17]
- [Adding, Removing and Saving Files](#)^[18]
- [Making, Uploading and Running an Executable Binary](#)^[26]
- [Debugging Your Project](#)^[28]
- [Project Settings](#)^[38]

The Structure of a Project

A *project* is a collection of related files and resources, which are then compiled into one final binary file, uploaded onto a target and run. It includes actual source files, HTML files (if any), images (if any), etc.

The actual parts of a project are:



Project file: A single file with a **.tpr** extension. Contains project settings, and a list of all files included with the project.



Header files: Multiple files with a **.tbh** extension. Used for inclusion into other files; usually contain declarations for global variables, constants, etc.



BASIC files: Multiple files with a **.tbs** extension. Contain the actual body of your program.



HTML files: Multiple files with an **.html** extension (displayed with the currently associated icon). Contain webpages to be displayed by the embedded webserver. These can include blocks of Tibbo Basic code. See [Working with HTML](#)⁷⁴.

(Any icon)

Resource files: Multiple files without any set extension. Contain resources (such as images) needed for other files. Some resource files (.cfg, .txt, .ini) can be edited from within TIDE:

- **.cfg**, **.txt**, and **.ini** files are considered to be text files and can be edited using TIDE's built-in text editor.
- **.bmp**, **.jpg**, and **.png** files are graphical files; these will be opened using TIDE's built-in [image editor](#)⁷⁵.



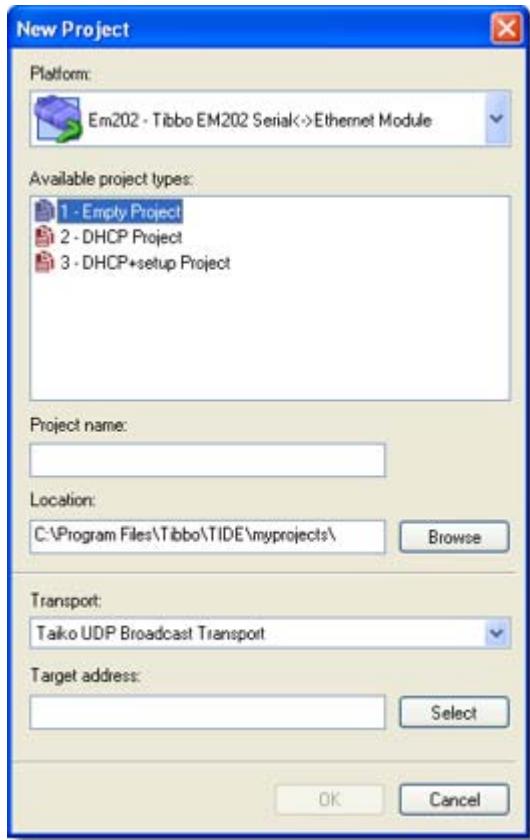
Note that the only really *set* extension is the one for the project file -- **tpr**. This file contains references to the other files within the project. These may use any extension, as long as their type is correctly stated in the project file (this is selected when adding the file, as described [here](#)¹⁸).

The extensions above are the default extensions which are associated with TIDE, and we recommend maintaining them.

The *project path* is a folder containing all the files described above.

Creating, Opening and Saving Projects

To create a new project, select File > New. The following dialog will appear:



Platform: The platform on which your project will run.

Available project types: Various quick-start [templates](#)¹⁷. Select the one which is closest to what you are trying to achieve, and use the resulting project as a basis for your work.

Project name: A short name for your project. TIDE will use this name to create a folder for your project, and also to create a project file (.tpr) within this folder.

Location: A folder containing all of your projects. This folder which will contain the project sub-folder. The actual files for your project will go into the project sub-folder.

Target address: *Platform-specific*. See [platform](#)¹³³ documentation. The address of the target you will use for debugging and testing this project. This should be a reachable address with a live target. Your project will still be created even if you do not specify this parameter, but you will not be able to upload or debug until you specify this setting using the [Project Settings](#)¹³⁸ dialog.

Templates.2.1

Templates are 'shell' projects. They contain the various operational parts which your project is likely to need. They are meant to be used as a starting point for creating complete projects.

Templates may contain header files, source files, and any other thing you may expect to find in a project. When you create a new project based on an existing

template, all files from the template folder are copied to your project folder, and the name of the project filename (the .tpr file) is changed according to your own project name.

Creating New Templates

It is also possible to create new templates for any platform. This is done by creating a template folder and populating it with files. You would then have to modify the platform file (.p file -- for example, em202.p) so that it would point to this template. This file lists the available template files for this platform, along with their paths (relative to the platform file).

For example:

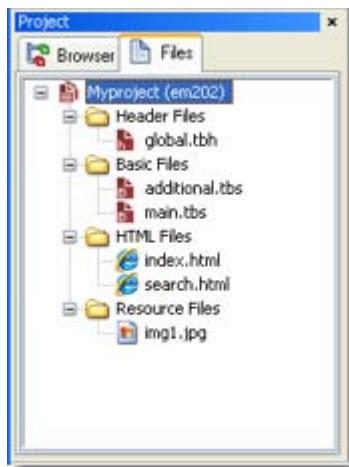
```
[template1]
name=Simple Project
description=Simple project template
icon=em202\simple.ico
path=em202\simple\simple.tpr
```

The definition above would specify a template project called Simple Project, whose path would be em202\simple\simple.tpr. Remember -- the folder and files for this project would not be automatically created.

Adding, Removing and Saving Files

Files tab

You can see what files are included in your project at any given time using the Files tab of the [Project pane](#)^[129]. It looks like this:

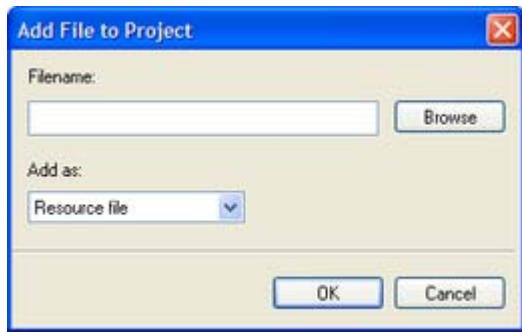


Adding New Files to Your Project



To add new files to your project, click Project > Add File or click the Add button on the Project Toolbar.

You will be presented with the following dialog:

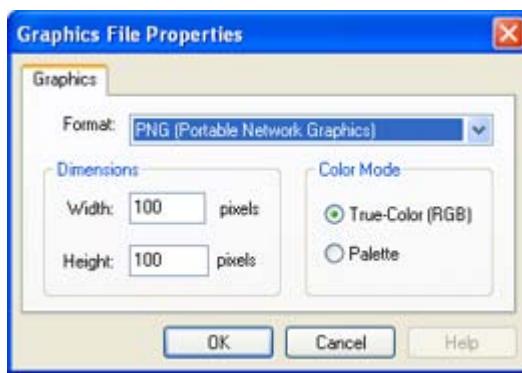


Specify a name for your file under Filename. If you also specify an extension, the Add as listbox is automatically updated. Automatically recognized file types are:

- *.tbs* -- basic files
- *.h* -- header files
- *.bmp, .jpg, .png, .gif, .ico, .pcx* -- graphic resource files
- *.txt* -- text resource files
- all other file extensions are classified as binary resource files by default.

File type is set according to the selection in the Add as listbox, not the file extension.

Adding [image](#)²⁰ files to the project (*.bmp, .jpg, .png, .gif, .ico, .pcx* extension) will prompt a request for additional information: file size in pixels and, if appropriate for the file type, the color mode selection (RGB or palette). This last selection won't be available if selected file type only supports true-color or palettized mode.



Format listbox will be disabled if the type of the image file you are adding is already known. Specifically:

- If your image file extension is not *.bmp, .jpg, .png, .gif, .ico*, or *.pcx*, then the listbox will be enabled.
- If you did not provide any extension at all, then the listbox will be disabled and the PNG type will be selected automatically.
- If you ended the filename with the comma (i.e. "abc.") then the listbox will be enabled.

Removing Files from Your Project



To remove a file from your project, first select (single-click) it in the project tree. Then click Project > Remove File or click the Remove button on the Project Toolbar.

You will be presented with a prompt. If you're sure you want to remove the file, select OK, and the file will be removed from the project. Note that it is not physically deleted -- only removed from the project tree.

Renaming Files

To rename a file, highlight it in the tree (single-click) and press F2, or single-click again.

Saving Files



To manually save your work, select File > Save, press Ctrl+S, or click the Save button on the Project Toolbar.

Any of these actions would save all open and modified files in your project, including the project file itself.

In addition, every time your project is compiled, all open and modified files are saved.

Resource Files

Sometimes a project may need access to certain files which are not Tibbo Basic code per se; these may be [image](#)²⁰ files, sound files or any other fixed binary data which is not to be interpreted or modified by the Tibbo Basic compiler but simply used as-is within the project.

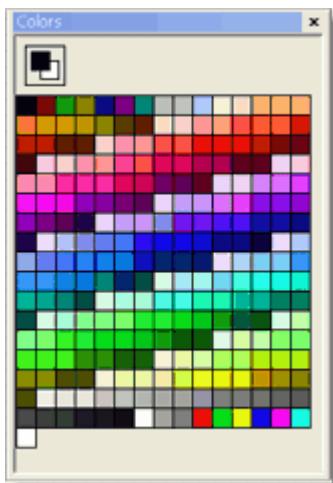
These files are not modified or compressed in any way; they are merely included within the final, compiled [binary file](#)²⁶ and may be accessed from within the program or by the built-in HTTP server.

Resource files are included in the [project tree](#)²⁹ under the Resource Files branch.

Built-in Image Editor

Beginning with release [2.0](#), TIDE features a simple image editor that "knows" how to work with *.bmp*, *.jpg*, *.png*, and *.gif* files. The editor is primarily intended for bitmap-level jobs such as preparing "screens" for a device with an LCD display.

All features of the image editor are fairly standard and we see no need in discussing image editor's functionality in details. Image files are [added](#)¹⁸ to the project as any other resource files. When adding an image file to the project you will be presented with a choice of selecting RGB or palette color mode for this file. Depending on your choice the [Colors pane](#)¹³⁰ for this image will display available palette colors...



... or RGB color selector:



Double-clicking on the image file with *.bmp*, *.jpg*, *.png*, or *.gif* extension opens the image editor.

All image-related editing functionality is concentrated within [Image menu](#)^[119]...



...and [Image Editor toolbar](#)^[122]:

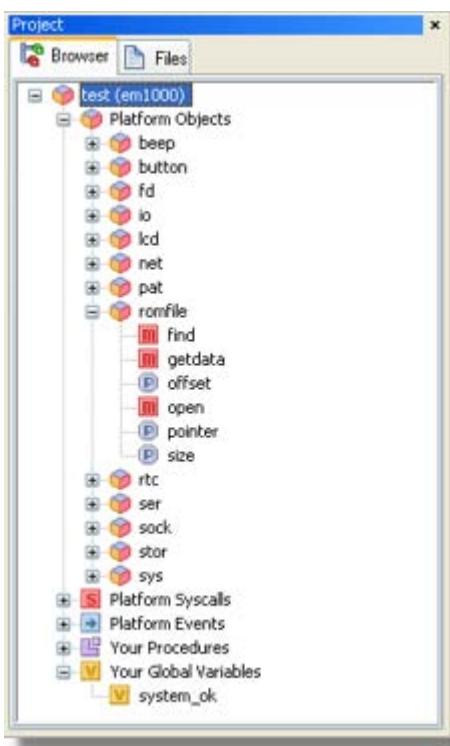


Coding Your Project

TIDE contains a code editor with the following facilities:

- Syntax highlighting
- [Auto-Completion](#)^[23]
- [Hinting](#)^[24]
- [Tooltips](#)^[24] for properties, events, functions, and even user-configurable tooltips for user-defined functions.

Project Browser



The Project Browser contains a tree of all objects in the platform (with their methods, properties and events), as well as all procedures and global variables of your project. The tree is updated in real time, using a dynamic background parser which constantly analyzes your source code.

The tree features [icons](#)^[13] for the various constructs.

An icon next to an event is grayed (inactive) if this event does not have an event handler implemented in the project. The icon becomes "active" when the event handler is created. An icon next to a procedure is grayed if this procedure is merely declared but does not yet have a body. The icon becomes active once the

procedure is implemented ("gets" a body). Same applies to global variables -- gray icon next to variables that are declared but not yet defined, active icon for defined global variables.

Double-clicking on an event which does not yet have an event handler will create an empty event handler procedure for this event at the bottom of the currently active file. Double clicking on an event which already has an event handler will make the cursor jump to this event handler.

Double-clicking on a procedure which does not yet have a body will make the cursor jump to the location where this procedure is defined. Double-clicking on a procedure which already has a body will make the cursor jump to this body.

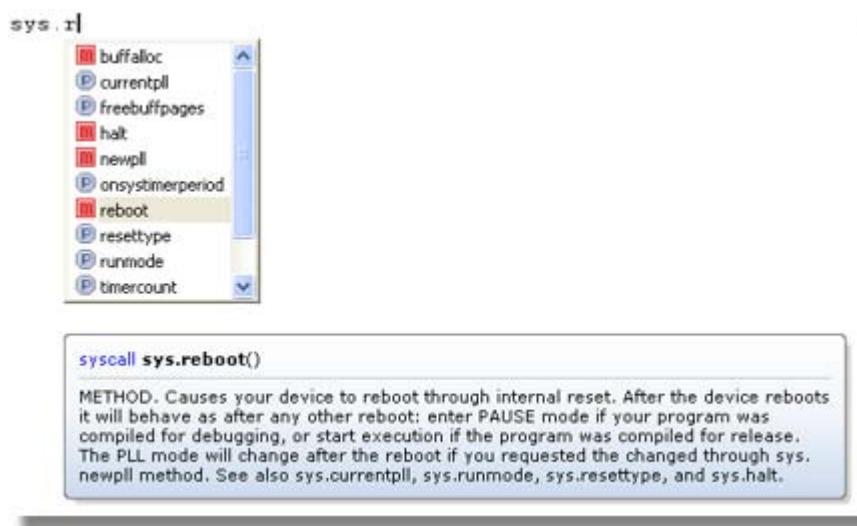
Double-clicking on a global variable which is not yet defined (using the [dim statement](#)) will make the cursor jump to the location where this variable is declared (using the [declare statement](#)). Double-clicking on a defined global variable will make the cursor jump to the location where this variable is defined.

Hovering the cursor over an item in the displays a [tooltip](#) for this item. Additionally, when in debug mode, hovering the cursor over global variables and object properties displays their current values.

Notice, that currently selected platform is displayed next to the project name in the topmost tree node.

Code Auto-completion

If you type an object name followed by a dot, the TIDE will pop-up a code-completion box. It looks like this:



This list also supports [tooltips](#) -- they are displayed as you scroll down the list. You can also hover the mouse over an item in the list to see its tooltip.

Code completion box is also displayed for the [structures](#) in your project:



You can also get a list of constants related to your construct. For example, by typing "ser.enabled=" you will get a list of possible values of this property:



By pressing Alt+Ctrl+T when the cursor is directly to the right of any meaningful Tibbo Basic construct, you will get a pop-up list with the appropriate contents for the current context. If the cursor isn't immediately to the right of any such construct, you will get a list containing all platform enumeration types, constants, objects and events.

Code Hinting

Code hinting is a feature which helps you see what are the arguments for a function, while writing the code for it. It appears as soon as you type the opening parentheses for a procedure. It looks like this:



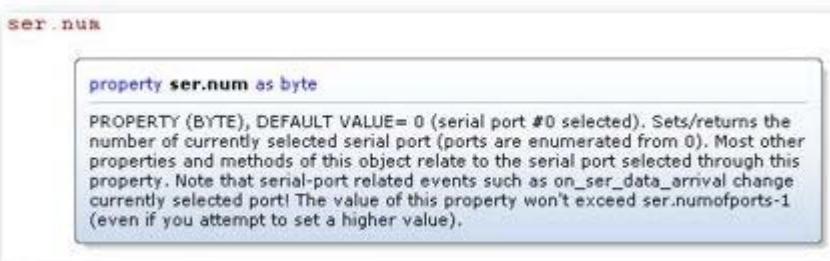
You can see the number of arguments required and their types, as well as the return type (if any). The highlighted part shows what syntax element you should type in next.



You may invoke code hinting manually by pressing Ctrl+Shift+space.

Tooltips

When you hover your mouse over event handlers, object properties and methods, constants, procedures, and variables the TIDE displays *tooltips*. These look like this:



The tooltips are displayed when hovering over constructs in the code editor, the [Project Browser](#)¹²⁹, [Watch](#)³³, and the [Stack](#)¹²⁸ pane. They show a formal construct definition and a comment, if available.



In the code editor, you may also display a tooltip with the keyboard by pressing **Ctrl+T** when the cursor is within an event handler, a procedure, a constant or a global variable.

Tooltip text for properties, methods, and events comes from the platform file for the [platform](#) selected in your project. You can add your own custom comments to the tooltips displayed for procedures and variables of your projects. What's more, you can use HTML formatting to make these comments look more readable! Here is an example:

```
function blink(num as integer) as boolean ' Blinks the lights.
Input:<br><b>num</b>- pattern # to "play". <b><font color=red> Do not set
to 0.</font></b>.

'... your code here ...

End Function
```

This would yield the following tooltip when hovering over the "blink" identifier (notice how HTML formatting improves tooltip readability):

[Supported HTML Tags](#)²⁶ section details which tags you can use to beautify your tooltips.

Your comment must be on the same line as the function definition, or immediately following it. The comment can contain multiple lines, if every line begins with a comment character. These lines must be consecutive -- with no blank lines in between. For example:

```
function blink(num as integer) as boolean ' USER-DEFINED. Blinks the
lights.
' This is a very important function.

' And must be included in every application.

'... your code here ...

End Function
```

This would yield a tooltip with the text "USER DEFINED. Blinks the lights. This is a very important function." This would be as one paragraph -- line breaks are not displayed within the tooltip, unless you use **
** element. The third comment would not be included because it is preceded by a blank line.

As we will explain in the [Introduction to Procedures](#)⁶², there is a procedure definition (procedure body) and procedure declaration that merely states that the procedure exists. If both have comments than the comment in the procedure

definition wins (takes precedence) over the comment in the procedure declaration.

Variables also have declaration ([declare](#)⁷⁹) [statement](#)⁷⁹) and definition ([dim](#)⁸¹) [s](#)⁸¹ [tatement](#)⁸¹). Comment in the definition wins.

Finally, your own comment placed in the event handler definition takes over the comment for this event that comes from the platform file.

Supported HTML Tags

Here are the tags (HTML elements) that you can use:

- Presentation markup tags: ``, `<i>`, `<big>`, `<small>`, `<s>`, `<u>`, `` [color=*color*] [size=*size*] [face=*face*]
- Headings `<h1>...<h6>`
- Line break `
`
- Comments: `<!-- -->`

All other tags (elements) cannot be used. Most of these tags are simply ignored, but some lead to scrambled text output.



One peculiarity of the HTML renderer used in the TIDE software is that it often requires you to add an extra space before the closing tag in the tag pair. For example, if you write "`Bold` text" then you will get this output: "**Boldtext**". Writing "`Bold ` text" or `Bold`text will produce correct result: "**Bold** text".

Making, Uploading and Running an Executable Binary

An *Executable Binary File* is a file (.tpc type) which contains your project in compiled form, along with any resource files. It is uploaded to the target, where it is actually executed by the TiOS Virtual Machine.

Making a Binary



Once you have code which you wish to try out, you may build it by selecting Project > Build, by pressing the shortcut key F7 or by clicking the Build button on the Debug toolbar.

If this is not the first time you're building this project, the build process will skip any files which were unmodified since the last time the project was built. This optimizes build speed.

To force the build of all files, even those which were not modified since the last time, select Project > Rebuild All.

On build, the [Output pane](#)¹²⁹ will display any errors. You can double-click on the line describing an error to jump directly to the problematic line in your code.

Uploading a Binary



To upload your project, you must select Project > Upload or click the Upload button on the Debug toolbar.

Before uploading, TIDE checks if the project has been changed since it was last built. If so, it builds the project again and attempts to upload the new build.

Also, the current project hosted on the target will be checked. If it is the same (same project and same build number) as the project you are trying to upload, uploading will not occur. Thus, trying to upload a project twice without making any change in the project will not result in a second upload. Also, before uploading, the firmware version is checked and if it is incompatible with the firmware version specified in the platform file, the upload is aborted with an error message.

Running a Binary

For a [debug version](#)^[38] (the default version type), uploading the binary does not automatically start its execution on the target. Once uploaded, it just sits there, waiting to be executed.



To begin execution, select Debug > Run, press the shortcut key F5 or click the Run button on the Debug toolbar.

This action optionally builds and uploads your application, if needed. If a new upload was just performed, it also *reboots* the target before running it. This ensures that target starts running the newly uploaded program from a 'fresh' state.



You may also reboot your device manually at any time by selecting Debug > Restart or clicking the Restart button on the Debug toolbar.



These actions are *incremental*. This means that when uploading, a build is performed if needed. When running, a build and an upload are performed if needed.

Two Modes of Target Execution

When you execute a program on the target, it can run in either of two modes (depending on the setting selected under [Project Settings](#))^[38]:

Debug Mode

In debug mode, your project runs with the assumption that you are right there, watching the monitor and trying to see what's going on. This means that the Debug menu is active. You can set up [breakpoints](#)^[30], or [step](#)^[32] through your project, watch the variables, etc.

This also means the project might stop if there's an error, such as division by 0. And you can pause execution, stop it, etc.

Also, when uploading a project in debug mode, it does not begin to run by default. By default, it waits for you to run it.



Selecting Debug > Run, pressing F5 or clicking the Run button on the Debug toolbar would send an explicit command to the target, to start running the project.

If the device reboots while a project is running in Debug Mode, the project will not start running automatically after the reboot. You would have to run it explicitly.

Release Mode

Release mode means business. This is the mode in which you compile the final files, deployed in the field. Under this mode, the working assumption is that you, or anybody else, isn't there. Your box is just supposed to run and run, despite any and all problems and errors.

This means that a release version does not respond to any debug commands. You cannot stop it. It does not stop even when critical errors occur. It also means that when you upload a release version to your target, it starts running immediately.

Even if you reboot your device, when it has a Release Mode binary in memory, it will start running.

Debugging Your Project

One of the most common operations you will perform during your development process is *debugging*. In essence, this involves controlled execution of your project. While debugging you can step through your program, set breakpoints, watch and change the state of various variables, see how control and decisions statements are executed, etc.

One of the aspects of TIDE is that it employs a technique called *cross-debugging*. Simply put, this means your code runs on a different machine than the one on which you wrote it, and you can debug it from the computer on which you wrote the program.

Thus, code is not debugged using some PC emulator or anything of this sort. It is truly uploaded and run on your target -- just like it would run in real life.

As covered [above](#)^[26], the first thing you would have to do to begin debugging a debug binary would be to run it, using F5. Once you press F5 (or Debug > Run), your project will be built (if necessary), uploaded (if necessary) and started.

Once execution has started, there are several ways in which you may control and inspect it. These are listed below.

Target States

In debug mode, your target may be in one of several states at any given moment.

For this, the [status bar](#)^[126] displays several different status messages:

RUN

Run: This message means your program is currently running. It doesn't mean any specific code is actually being executed -- perhaps the target device is just sitting idle, waiting for an event to happen. But the program is still running -- not paused. This state is entered by pressing F5 or Debug > Run.

BREAK

Break: This message occurs when the Virtual Machine on the target was stopped while executing code. The easiest way to get to this state is by setting and reaching a [breakpoint](#)^[30] in code. You might also get to this state by selecting Debug > Pause, if you happen to catch the Virtual Machine in the midst of code execution. Once in this state, the [program pointer](#)^[30] (a yellow line) is displayed and indicates the next instruction that the Virtual Machine will execute when started. You can now inspect and change various properties and variables (both global and local) using the [watch](#)^[33]. This is the only state which allows [stepping](#)^[32].

PAUSE

Pause: This message occurs when the Virtual Machine on the target was stopped while not executing code (in other words, it was caught between events). No program pointer is displayed in this state, because no code is being executed. You can check and modify the state of properties and global variables, etc using the watch. This state is entered by selecting Debug > Pause.

ABORT(DIV0)

Abort (exception): This message indicates that an internal error has occurred. The message in parentheses is a short error code. If you hover your mouse over it, you will see a more detailed report of the error. The program pointer will appear at the problematic line. This state is similar to a break, only it is not caused by a breakpoint but by an abnormal condition. All possible causes for exception are listed in [Exceptions](#)^[29].

Communication States

While TIDE is in communication with the target, the status bar displays a moving indicator of the communication state.



Communication in progress: The circle is green, and moves from side to side. It advances one step whenever TIDE gets a reply to a debug command.



Communication problem: The circle is yellow, and does not move. This state means TIDE did not receive any reply to debug commands for more than 6 seconds. The program may be still running on the target.



No Communication: The circle is red, and does not move. Occurs when TIDE did not receive any reply to debug commands for more than 12 seconds. The program may be still running on the target.

Exceptions

Exceptions are "emergency" halts of program execution. Exceptions are generated when the Virtual Machine encounters something that really prevents it from continuing normal operation. When exception happens you see "ABORT" target state in the status bar, like this:



"(DIV0)" is an abbreviated problem description. Hover the mouse over this and you will get a more detailed description.

Listed below are all possible exceptions. When you are in the [debug](#)^[27] mode any exception from the list below causes the Virtual Machine to abort execution. In the [release](#)^[27] mode, some "lesser" problems do not cause the halt. The logic here is that there will probably be nobody to restart the problem or check what happened anyway, so the Virtual Machine just tries to continue operation.

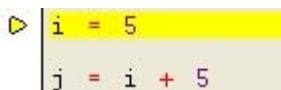
Cod e	Description	Halt in debug mode?	Halt in release mode?

DIV 0	Division by zero	Yes	No
OO R	Out of range (attempt to access past the highest array member)	Yes	No
FPE RR	Floating point error	Yes	No
IOC	Invalid opcode*	Yes	Yes
OU M	Access outside of user memory*	Yes	Yes
TDL F	Failed to load binary library*	Yes	Yes

*This exception indicates that either TiOS or Tibbo Basic compiler is not functioning properly. Let us know if you encounter this exception!

Program Pointer

The *program pointer* is a line, highlighted in yellow, which shows the present location of program execution. It looks like this:



This means that the yellow line is now pending execution. It hasn't been executed yet. The machine is waiting for you to tell it what to do. You can now control it by [stepping](#)^[32].

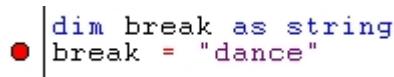
This line is displayed whenever the Virtual Machine has been paused while executing code. This can be achieved by setting a [breakpoint](#)^[30], or simply selecting Debug > Pause at the "right" time.

The program pointer will only stop on lines which contain *actual executable code*.

```
dim x as byte ' the program pointer won't stop here, as this isn't
executable code.
x = 1 ' the program pointer will stop here -- this is an actual
instruction to do something.
```

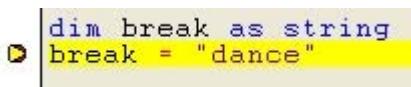
Breakpoints .3

A *breakpoint* is a point marking a line of code in which you wish to have the debugger pause. It is seen as a little red dot on the left margin of the code. Like this:



This is what it looks like when the code is not executing.

Once the code begins to execute and the breakpoint is reached, the [program pointer](#)^[30] is displayed at the line in which the breakpoint is placed:



The yellow arrow over the red dot merely marks the program pointer; a breakpoint is always marked by a red dot.

Where a Breakpoint May Be Placed

A breakpoint may be placed only on a line which contains executable code; before compiling your project, you could place breakpoints anywhere. However, on compile, these breakpoints will be shifted to the nearest lines following them which contain executable code.

You could add breakpoints to your code at any time -- even while the code is running. However, while the code is running, you may only add breakpoints next to lines which contain executable code. If you click next to a line which does not contain executable code, the closest line following this line which does contain executable code will get a breakpoint.

You may have up to 16 breakpoints in your entire project. Breakpoints are saved when the project is saved.



Adding breakpoints slows down the performance of the Virtual Machine. Having 16 breakpoints will have a noticeable effect on the speed of execution of your program.

Toggling Breakpoints

Breakpoints may be toggled (set/cleared) by putting the cursor in the line in which you wish to place (or remove) the breakpoint and pressing F9 or selecting Debug > Toggle Breakpoint. Alternatively, you may also toggle a breakpoint by clicking on the margin of the code at the point in which you wish to have a breakpoint.

You may remove all breakpoints from an entire project (including any files which are not currently open) by selecting Debug > Remove All Breakpoints.

The Call Stack and Stack Pointer

During the execution of a program, procedures usually call other procedures. The calling procedures are not just left and forgotten; they are placed on the *call stack*.

The call stack can be toggled by View > Call Stack. It lists the sequence of procedure calls which lead to the current procedure (the one in which the [Program Pointer](#) is located). The current procedure is listed at the top of the stack. Once it finishes executing, execution returns to the caller (the procedure one line below in the stack). In the caller procedure, execution resumes from the line immediately following the one which called the procedure which has just ended.

So, if the procedure (event handler, actually) **on_timer** called the procedure **error_handler**, which in turn called the procedure **write_logfile**, our call stack would look like this:



When pausing a program in the midst of code execution, the [program pointer](#)³⁰ appears. In the Call Stack pane, the function which currently contains the program pointer is highlighted in yellow. It is the currently executing function, so it is always the first one on the Call Stack list.



Technically speaking, the top function on the call stack isn't actually a part of the *stack* itself, because it is currently executing, and the real stack only contains functions to which execution should later return. It still appears on the same list, for consistency and convenience.

The Stack Pointer

Double-clicking on any procedure within the call stack which is not the currently executing procedure would display the *stack pointer*. This pointer would be displayed in the source code, within the procedure you double-clicked, and would highlight the line from which execution would resume once control returns to this procedure. The [watch pane](#)³³ would also interpret variables as relative to the procedure you've just highlighted.

The stack pointer is light blue in color. On the call stack list, it looks like this:



In the code editing pane, it looks like this:

```
sub error_handler
dim x as integer
x = x + 5
write_logfile
end sub
```

Once again, double-clicking on the functions in the call stack *does not move actual execution* (the program pointer). Any sort of stepping would bring back the yellow program pointer, both in the source code and in the call stack.

Stepping 1.8.5

The following commands in the Debug menu control stepping into, through and out of various sections of your code:

	Step Into	F8,F11
	Step Over	F10
	Step Out	Shift+F11
	Run to Cursor	Ctrl+F10
	Jump to Cursor	Ctrl+F11

A *step* is an instruction to move the [program pointer](#)³⁰ to another line in source code. Stepping allows you to execute your code in a very controlled way, and work your way along the program in a pace which you can analyze and understand.



Step Into: Steps through your code line by line. When the program pointer reaches a procedure call, you would actually see it step *into* this procedure (hence, the name). You could then see the inner workings of this procedure as it is being executed, line by line.



Step Over: Steps through your code line by line. When the program pointer reaches a procedure call, it executes this entire procedure, but just does it all at once, and stops at the next line after the procedure call. This is useful when you want to debug a body of code which contains a call to a complex or lengthy procedure, which you do not want to debug right now.



Step Out: If you are currently stepping through a function and wish to exit it while you're still in the middle, use Step Out. This would bring you to the line immediately following the line which called the function you were in. This option is disabled when you cannot step out of the current function (i.e., when your other function calls it -- such as in the case of an [event handler](#)⁸).



Run to Cursor: The *cursor*, in this case, is the text insertion point. The little blinking black line. You can place the cursor anywhere within the body of your program and have the program execute until it reaches that point. When, and if, that point is reached, the program pointer would display.



Jump to Cursor: This command makes the program pointer unconditionally move to the point where the cursor is. It will just *jump* there, possibly skipping the execution of some code. This is explicit control of program flow.

The Watch^{8.6}

The *watch* is a facility which allows you to inspect and change the current value of variables and object properties. You can only use this facility when you are in debug mode and when the program execution is stopped. TIDE is unable to fetch variable values while the Virtual Machine is running. Depending on the scope of the variable, there may be [additional limitations](#)³⁷ as to when you can inspect this variable's value.

The watch updates variable values by reading them from the target every time the Virtual Machine is stopped. If any item on the list changed its value since the previous fetch, this item will be displayed in red.

Watch facility may be accessed by three different ways:

The Watch Pane

The watch can be toggled by View > Watch. It looks like this:

Name	Type	Value
sh1	short	-100 (&hff9c)
mystr	string(255)	12: "Test string." (&h54 65 73 74 20 73 74 72 69 6e 67 2e)
sys.runmode	pl_sys_mode	0 - PL_SYS_MODE_RELEASE
x(y)	short	20 (&h0014)
struct1	my_struct	(struct)
r1	real	1.2e-007
flag	no_yes	1 - YES
s	string(255)	4: "TEST" (&h54 45 53 54)
xxx	?	?

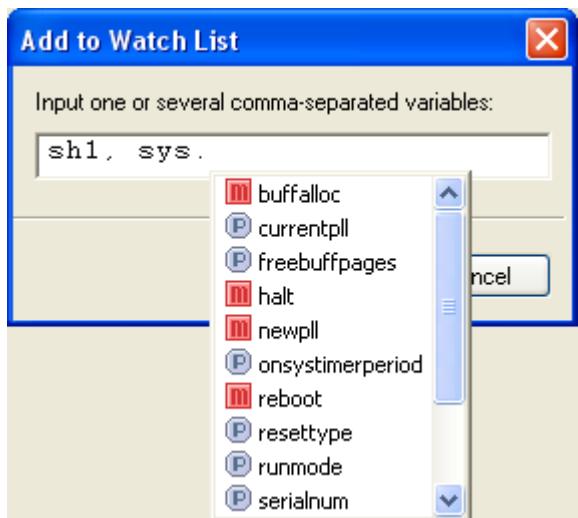
On the above screenshot we can see the status of a **short**, a **string**, a **property** of an object, a member of an **array**, one user-defined **structure**, and an undefined **variable**. Notice, that the "x(y)" is displayed in red indicating that its value has just changed. Notice also that this line shows an array indexed by another variable (y). This is not the limit of the watch pane's abilities -- try entering more complex expressions, it will work!



There are several ways of adding variables and object properties to the watch list. You can:

- Press Insert while the watch pane has focus -- this will bring up an Add to Watch List dialog. Type the name of a variable or property to watch and press OK.
- Alternatively, double-click on the empty space in the watch pane to obtain the same result.
- You can also select Debug > Add to Watch List from the Menu.
- Additionally, there is an Add to Watch List button on the Debug toolbar.
- You can right-click on the variable or property in the source code and select Add to Watch List from the context-sensitive menu.
- You can also right-click on the property in the Project pane (Browser tab) and select Add to Watch List.

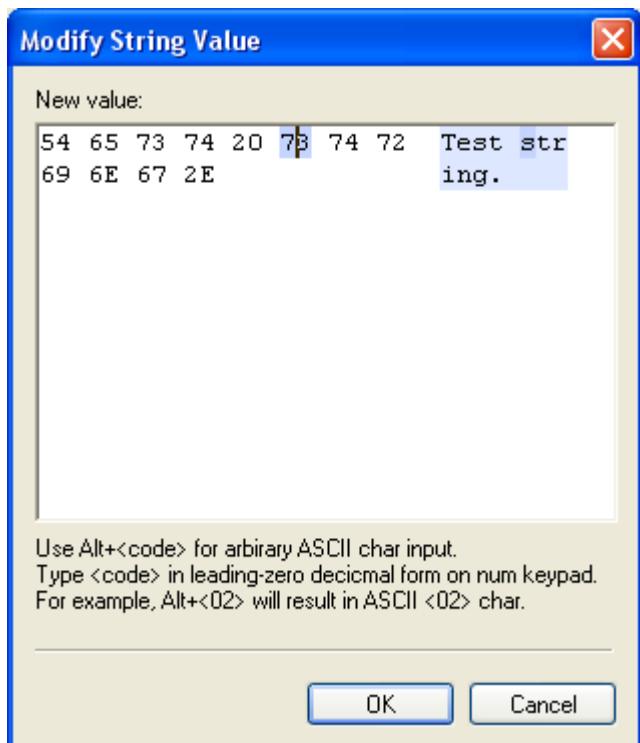
In the Add to Watch List dialog you may type multiple items to be added to the watch by separating their names with commas (i.e. "i, j, k, ser.num"). For objects you will get a drop-down list of available members:



Double-clicking the Name column in the watch pane will allow you to edit the name of the item you want to watch.

The watch pane also allows you to change the value of any variable or object property (provided it is not a read-only property). Double-click on the value field - you will be prompted for a new value.

For numerical variables, you may use hex or binary notation (&h, &b). For strings, you will be presented with the "HEX editor" allowing you to modify the string or the HEX codes of its characters:



Once a new value is entered, it will be actually written to memory on the target, and will be read again before being displayed in the watch. So when you see your new value in the value column, that means it's actually in memory now -- what

you see is what you get.

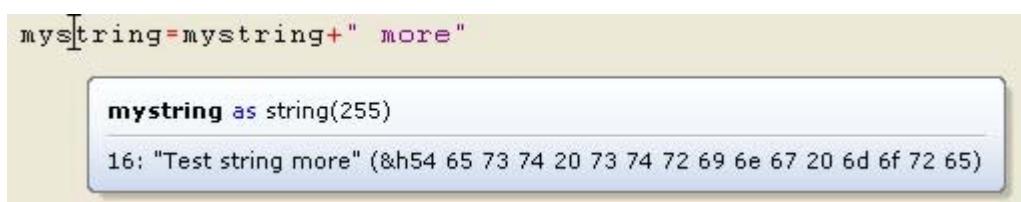
The watch pane is one of the places where enumeration types become very useful. Look at sys.runmode above. Because its possible values are described through an enum, you can see a meaningful state description, rather than just a number. This is one of the main reasons for the existence of enumeration types in Tibbo Basic.



To remove a variable from the watch, select it and press Delete, select Debug > Remove from Watch List or click the Remove from Watch List button on the Debug toolbar.

The Watch Tooltip

This is the watch tooltip:

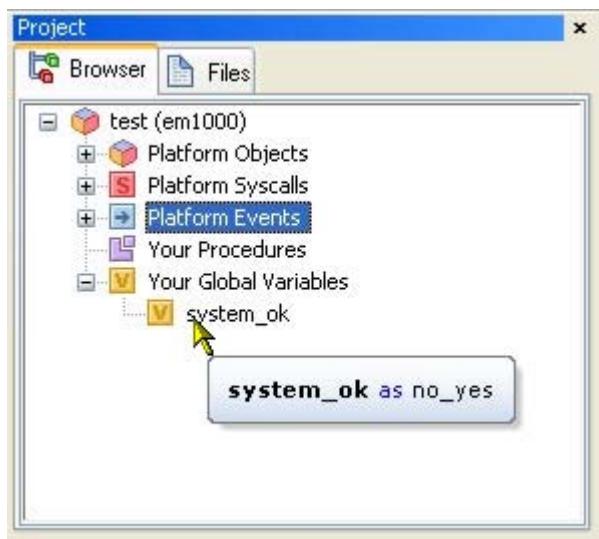


When you hover the mouse cursor over an identifier in the source while in debug mode you will see a tooltip with the current value for this identifier.



For now, this won't work for arrays. If you want to inspect the values of an array add this array to the watch list.

The Project Browser



The Project Browser is a tab in the Project pane, which can be toggled by View > Project. It displays all objects for your platform, with the properties and methods for each object. It also displays all procedures and global variables in your project.

While debugging, you can see the value of an object property or a global variable by hovering over its identifier in the tree.

Scopes in Watch

The watch facility is only active when the Virtual Machine is not running, i.e. the execution is stopped. Naturally, the TIDE cannot fetch variable values while the Virtual Machine is executing your program.

You already know that when the Virtual Machine it stopped, the [state of your target](#)²⁸ may be either "BREAK" or "PAUSE". Properties and global variables may be inspected in the either state. Local variables only exist in their *context*. Hence, a particular local variable can only be inspected when the state is "BREAK" (the Virtual Machine is in the middle of a code execution -- there is an execution pointer visible) and when this variable exists in the *current context*.

For example:

```
sub sub_one
    dim x as byte
    x = 1
end sub

sub sub_two
    dim x as byte
    x = 2
end sub

sub sub_three
    dim i as integer
    i = 5
end sub
```

Let us say you add x to the watch list.

When the execution pointer highlights the **end sub** keyword for sub_one, the value of x in the watch would be 1. When the pointer highlights the **end sub** for sub_two, the value of x in the watch would be 2. Note that these are two *different local variables!*

Similarly, when the execution pointer is at the **end sub** of sub_three, the x variable in the watch will be undefined -- it will display a question mark.

These same rules apply to the watch tooltip, as well. Even if you hover the mouse over the x of sub_one when the program pointer is at the end of sub_two, the value displayed would be the one set in sub_two -- because this is the current context!

Code Profiling

Code Profiling is the practice of measuring how long it takes for different portions of your program to execute.

0.01 sec

This is the timer, located on the Status Bar.

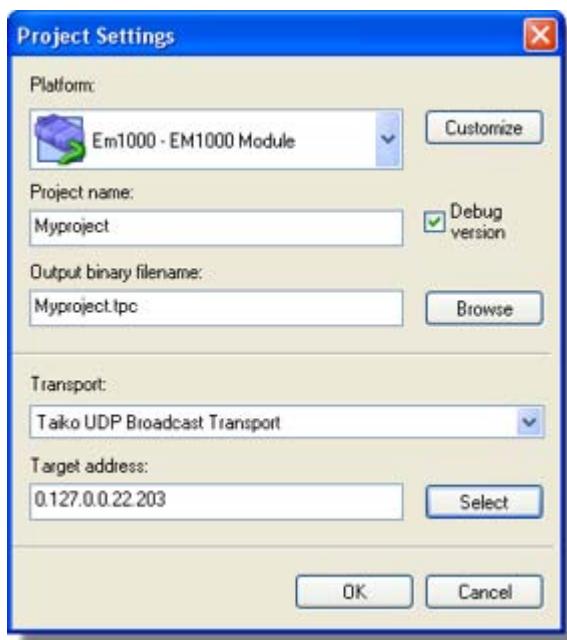
Each time execution begins or resumes in debug mode, the timer is reset and starts counting. Once execution is stopped for whatever reason, the timer displays the time elapsed since execution has begun or resumed.



Remember, as covered [above](#)³⁰, every additional breakpoint somewhat slows down the speed of execution of your project.

Project Settings

The Project-Settings dialog is platform specific. It can look like this:



Platform: This is the platform definition file. In the image above, EM202 is the platform.

Customize: click to access the list of option available for your platform. These are "global defines" (for a [preprocessor](#)) , which typically include options such as whether a display is present, display type, etc.

Project name: A descriptive name for the project.

Debug version: If checked, the compiler will build a version of this project which can be debugged using the various debugging facilities within the IDE, such as step, watch, etc. Also, debug versions are not automatically run on upload or reboot; release versions are run on upload or reboot, and cannot be debugged using the various debug facilities. By default, this is checked. Once you've debugged your project and wish to deploy it, uncheck this and build your final version.

Output binary file name: The name for the bin file of the compiled project.

Transport: Ethernet is not the only potential means of communications between the TIDE and the target. Depending on the hardware you are programming for there may be no Ethernet interface at all. This selector allows you to choose an alternative transport protocol for debug communications.

Target address: *Platform specific.* Different platforms use different communication media between TIDE and target -- TCP/IP, serial, etc. Thus, their target address may not always be a MAC address. This goes for the **browse** button as well -- not all platforms have this button.

Programming Fundamentals

This chapter attempts to provide a very quick run through the fundamentals of creating a program in Tibbo Basic. It was written under the assumption that the reader has some experience in programming for other languages.

There is a marked resemblance between Tibbo Basic and other types of BASIC that you may already know. Thus, we stressed the differences between Tibbo Basic and other BASIC implementations. Some sections begin with a seemingly introductory statement, but the material quickly escalates into more advanced explanations and examples.

This is not a programming tutorial. We do not attempt to teach you how to program in general. There are many excellent books which already exist on this subject, and we did not set out to compete with any of them. This is a mere attempt at explaining Tibbo Basic -- no more, no less.

Good luck!

Program Structure

A typical Tibbo Basic source code file (a [compilation unit](#) [13]) with the [.tbs](#) [15] extension) contains the following sections of code:

Include Statement(s)

These are used to [include](#) [90] other files from the same project (such as header files [.tbh](#) [15]) containing global variable definitions or utility functions). See:

```
include "global.tbh"
```

Global Variables Definitions

Here you define any variables you wish to be accessible throughout the current compilation unit:

```
dim foo, bar as integer
```

Subs and Functions

These are procedures which perform specific tasks, and may be called from other places within the project.

```
sub foo
...
end sub

function bar (a as integer) as byte
...
end function
```

Event Handlers

Tibbo Basic is [event-driven](#) [8]. Event handlers are platform-dependent subs, that are executed when something happens. This 'something' depends upon your

platform. If your Tibbo Basic program runs on a refrigerator, you would probably have an event handler for the door opening.

```
sub on_door_open
    light = 1 ' turn on the light when someone opens the door to your
fridge.
end sub
```

Just like any other sub, events can have arguments (input parameters):

```
sub on_door_state(state as byte)
    ' turn the light on and off as the fridge door gets opened and closed
    if state=0 then
        light=0
    else
        light=1
    end if
end sub
```

Event handles are always subs and never functions (i.e. they never return any value as there is nobody to return it to)!

Code Basics

There are several important things you should know about writing in Tibbo Basic:

You Can Put Comments in Your Code

The apostrophe character marks the beginning of a comment. Anything following this character until the end of a line is considered to be a comment, and will not be processed by the [compiler](#)^[13].

```
x = 1 + 1 ' I am a comment!
' x = y/0 <--- this line would not cause an error, because it won't even
execute! It is commented.
```

The only exception to this is that when including an apostrophe within a string (between double quotes) it is not counted as a comment. See:

```
s = "That's a string!" ' Notice that the word that's contains an
apostrophe.
```

Comments cannot span multiple lines. A line break terminates a comment. If you want to make a multi-line comment, each line of your comment must begin with an apostrophe.

Tibbo Basic Doesn't Care About Spaces!

See:

```
y =      x      +      5 ' is just like
y=x+5 ' and even this is OK:
y =
x
+
5
```

So, spaces, tabs and linefeeds carry no special meaning in Tibbo Basic. Use them or lose them, as you like. The only exceptions are the [If.... Then... Else Statement](#) ⁸⁹ and comments.

Tibbo Basic Is Not Case Sensitive!

See:

```
Z = X + Y ' is just like
z = x + y
r = Q + KeWL ' even something like this is legal.
Dim MooMoo As iNTeGER ' this, distatesful as it may be, is still legal. :)
```

Capital letters just don't matter. Really.

How to Use Double Quote Marks

Double quote marks are used for marking string literals. Simply put, a *string literal* is a constant string value -- like "hello world".

```
s = "I am a string literal!"
s1 = s
```

How to Use Single Quote Marks

These are different than the apostrophes which begin a comment. An apostrophe looks like this ' while a single quote mark looks like this ` and is usually located on the tilde (~) key. Single quote marks are used to define a numerical constant which contains the value for an ASCII code. For example:

```
dim b as byte
dim w as word
b = `q` ' the variable b now contains the value 113, which is the
character code for q.
w = `LM` ' this is also legal, see below.
```

The notation used in the second example above actually places the ASCII value of L into the higher byte of a [word-type](#) ⁴³ variable, and the ASCII value of M into the lower byte of that variable. It may seem confusing at first, but it is also legal.



Note that a this isn't the same character as an apostrophe. An apostrophe is ' and a single quote-mark is a ` . Usually, the single quote mark is found on the tilde (~) key, and the apostrophe is found on the double-quote (") key.

How to Define Constants In Different Bases

Tibbo Basic allows you to assign values to constants using decimal, hexadecimal or binary notation. Decimal notation is the default. To assign a hexadecimal value, you must use the prefix **&h** before the first hexadecimal digit of your value. To assign a binary value, you must use the prefix **&b** in the same manner. So:

```
q = 15 ' this is 15, in decimal.
q = &hF ' this is still 15, just in hex.
q = &b1111 ' and this is also 15, just in binary notation.
```

These prefixes hold true whenever values are used -- there are no exceptions to this rule. Whenever a numeric value is used, it may be preceded by one of these prefixes and will be interpreted correctly.

How to Use Colons

Colons are actually not necessary in most parts of Tibbo Basic. They are a traditional part of many BASIC implementations, and are often used to group several statements in one line. However, since Tibbo Basic doesn't really care about spaces anyway, they lose their relevance.

No Tibbo Basic statements require the use of colons.

Naming Conventions

Identifiers

An identifier is the 'name' of a constant, a procedure or a variable. It is case insensitive. It may include letters (A-Z, a-z), digits (0-9) and the following special characters: . ~ \$!. It must start with a letter, and cannot contain spaces.

For example:

```
dim a123 as integer ' A legal example
dim 123a as integer ' An illegal example
dim a.something as integer ' Can contain dots.
sub my_sub (ARG1 as integer, ARG2 as string) ' This is also legal
```

Platform Objects, their Properties and Methods

The name of an object is used as a prefix, followed by a dot, followed by the name of the property or the method you would like to access. For example:

```
ser.send ' Addressing the 'send' method of a 'ser' object.
x = ser.baudrate ' Assigning variable X with the value of the 'baudrate'
of a 'ser' object
```

Events

Events are related to specific objects, just like properties and methods. However, events are named differently. The pattern is `on_objectname_eventname`. Such as `on_door_open`.

```
sub on_ser_data_arrival ' event names may contain more than one word after  
the object name.
```

Introduction to Variables, Constants and Scopes

Variables and constants are a major part of any programming language, and Tibbo Basic is no exception; below you will find explanations on the following:

- [Variables And Their Types](#)⁴³
- [Type Conversion](#)⁴⁵
- [Type Conversion in Expressions](#)⁴⁸
- [Compile-time Calculations](#)⁴⁹
- [Arrays](#)⁵⁰
- [Structures](#)⁵⁴
- [EnumerationTypes](#)⁵⁵
- [Understanding the Scope of Variables](#)⁵⁷
- [Declaring variables](#)⁶⁰
- [Constants](#)⁶⁰

Variables And Their Types

Variables are used to store values during the execution of an application. Each variable has a *name* (the [identifier](#)¹³² used to refer to the value the variable contains) and a *type*, which specifies how much data this variable can contain, and also what kind of data it may contain.



Not every variable type is supported on every platform. You will find related information in the "Supported Variable Types" topic in the platform documentation. If you attempt to use a type which is not supported by your platform you will most probably get "platform does not export XXX function" error during compilation.

Variables are defined using the [Dim Statement](#)⁸¹ prior to being used in code. The simplest syntax for defining a variable would be something like:

```
dim x as integer ' x is a variable of type 'integer'.  
dim str as string(32) ' str is a variable of type 'string' with a maximum  
length of 32 characters (bytes).
```

A variable name must begin with a letter, and must be unique within the same scope, which is the range from which the variable can be referenced.

When you define a variable, some space is reserved for it in memory; later, while the program executes, this memory space can hold a value.

Variables are assigned values like so:

```
x = 15 ' x is now 15.
x = y ' x is now equal to y.
str = "foobar" ' str now contains the string 'foobar' (with no quotes).
```

Types of Variables

Tibbo Basic supports the following variable types:

Name	Description
byte	Hardware-level. Unsigned. Takes 1 byte in memory. Can hold integer numerical values from 0 to 255 (&hFF).
word	Hardware-level. Unsigned. Takes 2 bytes in memory. Can hold integer numerical values from 0 to 65535 (&hFFFF).
dword (new in V2.0, not available on all platforms)	Hardware-level. Unsigned. Takes 4 bytes in memory. Can hold integer numerical values from 0 to 4294967295 (&hFFFFFFF).
char	Hardware-level. Signed. Takes 1 byte in memory. Can hold integer numerical values from -128 to 127.
short	Hardware-level. Signed. Takes 2 bytes in memory. Can hold integer numerical values from -32768 to 32767.
integer	Compiler-level. Synonym for short ; substituted for short at compilation. Exists for compatibility with other BASIC implementations.
long (new in V2.0, not available on all platforms)	Hardware-level. Signed. Takes 4 bytes in memory. Can hold integer numerical values from -2147483648 to 2147483647
real (new in V2.0, not available on all platforms)	Hardware-level. Signed, in standard "IEEE" floating-point format. Can hold integer and fractional numerical values from +/- 1.175494E-38 to +/- 3.402823E+38. Real calculations are intrinsically imprecise. Result of floating-point calculations may also differ slightly on different computing platforms. Additionally, floating-point calculations can lead to floating-point errors: #INF, -#INF, #NaN. In the debug mode, any such error causes an FPERR exception .
float (new in V2.0, not available on all platforms)	Compiler-level. Synonym for real ; substituted for real at compilation. Exists for compatibility with other BASIC implementations.

string	Hardware-level. Takes up to 257 bytes in memory (max string size can be defined separately for each string variable). Strings can actually be up to 255 bytes long but always begin with a 2-byte header -- 1 byte specifies current length, and 1 byte specifies maximum length. Each character is encoded using an ASCII (single-byte) code.
boolean	Compiler-level. Intended to contain one of two possible values (true or false). Substituted for byte at compilation.
user-defined structures (new in V2.0)	Each user-defined structure can include several <i>member variables</i> of different types. More about structures here ⁵⁴ .
user-defined enumeration types	Compiler-level. These are additional, user-defined, data types. More about these under User-Defined Types ⁵⁵ .



Hardware-level types are actually implemented on the machine which is used to run the final program produced by the compiler.

Compiler-level types are substituted by other variable types on compile-time. The actual machine uses other variable types to represent them; they are implemented for convenience while programming.

Type Conversion

Variables can derive their value from other variables; in other words, you can assign a variable to another variable. A simple example of this would be:

```
dim x, y as byte
x = 5 ' x is now 5
y = x ' y is now 5 as well
```

However, as covered above, there are several types of variables, and not all of them can handle the same data. For example, what would happen if you assigned a variable of type **byte** the value intended for a variable of type **word**?

Table below details all possible conversion situations.

	C o n v e r t i n t o
--	-------------------------

		Byte	Word	Dword	Char	Short	Long	Real	String
Convert from	Byte	---	OK	OK	Reinterpret	OK	OK	OK	OK str [207]
	Word	Truncate	---	OK	Reinterpret Truncate	Reinterpret	OK	OK	OK str [207]
	Dword	Truncate	Truncate	---	Truncate	Truncate	Reinterpret	OK	OK Istr [199]
	Char	Reinterpret	Reinterpret	Reinterpret	---	OK	OK	OK	OK str [208]
	Short	Reinterpret Truncate	Reinterpret	Reinterpret	Truncate	---	OK	OK	OK str [208]
	Long	Reinterpret Truncate	Reinterpret	Truncate	Truncate	Truncate	---	OK	OK Istr [200]
	Real	Fract ion ???	Fract on ???	Fract on ???	Fract on ???	Fract on ???	Fract on ????	---	OK fstr*
	String	OK val [210]	OK val [210]	OK lval [200]	OK val [210]	OK val [210]	OK lval [200]	OK str tof [209]	---

*fstr is a functional equivalent of [ftosstr](#)[194], but without mode and rnd parameters.

Conversions without loss

Conversions marked as "OK" incur no loss -- the value being passed from variable of one type to variable of another type remains unchanged. For example, conversion from **word** into **dword** is done without any loss, because 32-bit **word** variable can hold any value that the 16-bit **word** variable can hold.

Conversions that cause reinterpretation

Conversions marked with "Reinterpret" mean that although binary data held by the receiving variable may be the same this binary variable *may* be interpreted differently on the destination "side". As an example, see this conversion from **byte** into **char**:

```
dim x as byte
dim c as char
x = 254
c = x ' c now contains the binary value of 254, which is interpreted as -2
```

In the above example, both x and c will contain the same binary data. However, c is a signed 8-bit value, so binary contents of 254 mean -2. Strictly speaking, this reinterpretation will only happen if the value of x exceeds maximum positive number that c can hold -- 127. If $x \leq 127$ conversion will not cause reinterpretation. For example, if $x=15$ then doing $c=x$ will result in $c=15$ as well.

In fact, in some cases, conversion from unsigned type to a signed time will never result in the reinterpretation. This is when the maximum value that the source unsigned variable can hold can always fit in the range of positive values that the signed destination variable can hold. Example: conversion from byte (value range 0-255) to short (value range -32768 to 32767) will never result in the reinterpretation.

Conversion from signed type into unsigned type will always cause reinterpretation if the source variable contained a negative value.

Conversions that cause truncation

Conversions marked with "**Truncate**" mean that part of the binary data (on the most significant side) *may* be lost during the conversion. For example, converting from **word** type into **byte** type will only leave 8 bits of the original 16-bit value:

```
dim x as byte
dim w as word
w = 12345 'hex representation of 12345 is 3039
x = w ' now x contains 57. Why? Because only '39' of '3039' could fit in,
and decimal of &h39 is 57.
```

Notice, that some conversions will cause reinterpretation and truncation at the same time!

Conversions that round the number (remove fractions)

Conversions from real type into any other numerical type will cut off the fraction, as real is the only type that can hold fractions. Such conversions are marked as "**Fraction**" in the table above.

Conversions that implicitly invoke functions available to the user

Some conversions automatically invoke functions (syscalls) available for use in your program. In such cases the table above lists the name of the function invoked. For example, conversion from byte into string relies on the [str](#)^[207] function. Two ways of conversion below produce identical result:

```
dim x as byte
dim s as string
s = str(x) ' explicit invocation
s = x ' implicit invocation. Compiler is smart enough to use str for this
conversion
```

Conversion of Boolean Variables

Boolean variables are actually stored as **byte** type variables; thus, all notes above for **byte** type variables hold true for **boolean** variables as well.

Conversion of Enumeration Types

User-defined enumeration types are held in various variable types, depending on the values associated with the constants within the enumeration type. This is described in detail under [User-Defined Types](#). Thus, they are converted according to the variable type used to store them (described above).

Type conversion in expressions

In the [Type Conversion](#) we already explained what happens when you assign the value of a variable of one type to a variable of another type. This section deals the cases where variables of different types are used in expression. For example, if x is **byte** and i is **integer**, what will happen when you do "x+i"?

The Virtual Machine operates on 16-bit values by default

Native data width for the Virtual Machine is 16 bits. When you are performing calculations on 8-bit and/or 16-bit variables, result is always truncated to 16 bits. Also, all intermediary calculations are done using 16-bit arithmetic. Consider the following example first:

```
dim x,y,z as byte
x=150
y=200
z=(x+y)/10 ' result of 35 is within byte variable's range, but
intermediary calculations require 16-bit arithmetic.
```

The above example will give correct result. Even though all three variables are of **byte** type, internal calculations are 16-bit, so when x+y produces 350, which is beyond the range of the **byte** variable, the Virtual Machine handles this in the right way. Now, let's see another example:

```
dim i,j,k as word
i=50000
j=60000
k=(i+j)/10 ' result will be 4446, which is incorrect. 32-bit arithmetic
was not automatically used!
```

This example requires 32-bit calculations because i+j will produce a number which is outside the scope of 16-bit calculations. However, our compiler will not invoke 32-bit calculations automatically. Read on, this is not all...

Mixing in a 32-variable will cause the compiler to use 32-bit calculations

For the example above, turn i or j into a dword. Now, your calculations will be correct. Mixing in any 32-bit variable will automatically upgrade calculations to 32 bits!

```
dim i,k as word
dim d as dword
i=50000
d=60000
k=(i+d)/10 ' result will be 11000. This is correct!
```

String and non-string variables cannot be mixed!

Yes, sorry, but you cannot do the following (compiler will generate [type mismatch](#) 104 error):

```
'Wrong!!!
dim x as byte
x=5+"6"
```

In case you are wondering why `x="6"` would work but `x=5+"6"` doesn't: the latter is a mixed expression in which the compiler cannot decide what is implied: conversion of string to value and then addition, or conversion of value to string and then string concatenation!

Compile-time Calculations

Tibbo Basic always tries to pre-calculate everything that can be pre-calculated during compilation. For example, if you write the following code:

```
dim x,y as byte
y=5
x=5+10+y 'compiler will precalculate 5+10 and then the Virtual Machine
will only have to do 15+y
```

Pre-calculation reduces program size and speeds up execution.

When processing a string like "`x=50000`" compiler first determines the variable of what type would be necessary to hold the fixed value and chooses the smallest sufficient variable type. For example, for 50000 it is, obviously, **word**. Next, compiler applies the same rules of [type conversion](#) 45 as between two actual variables. Hence, in our example this will be like performing **byte= word**.

One additional detail. Large fixed values assigned to variables of real type must be written with fractional part, even if this fractional part is 0. Consider the following example:

```
dim r as real
r=12345678901 'try to compile this and you will get "constant too big"
error.
```

Compiler will notice that this constant does not fit even into **dword** and will generate an error. Now, try this:

```
dim r as real
r=12345678901.0 'that will work!
```

On seeing ".0" at the end of the value compiler will realize that the value must be treated as a **real** one (floating-point format) and process the value correctly.

So, why is it possible for compiler to automatically process values that fit into 8, 16, and 32 bits, but at the same time requires your conscious effort to specify that the value needs to be treated as a floating-point one?

This is because floating-point calculations are imprecise. The value "12345678901", when converted into a floating-point format, will not be exactly the same! The floating-point value will only *approximate* the value we intended to have!

For this reason we require you, the programmer, to make a conscious choice when specifying such values. By adding ".0" you acknowledge that you understand potential imprecision of the result.

Arrays 4.2.4.5

An *array* is a single variable which contains several elements of the same type. Each element has a value and an index number, and may be accessed using this number.

An example of a simple array would be:

Index:	0	1	2	3	4
Value:	15	32	4	100	-30

The code to produce such an array in Tibbo Basic would look like this:

```
dim x(5) as char
x(0) = 15
x(1) = 32
x(2) = 4
x(3) = 100
x(4) = -30
```

The first index in an array is **0**. Thus, the array above contains 5 values, with indices from 0 to 4.



This is different from some BASIC implementations that understand x(5) as "array x with the maximum element number of 5" (that is, with 6 elements). In Tibbo Basic declaring x(5) means "array of 5 elements, with indices from 0 to 4".

Variable Types For Arrays

In the example above, the array was assigned the type **char**. This means that each element within this array will be stored in a variable of type **char**⁴³. Starting from Tibbo Basic **V2.0** you can have arrays of variables of any type.

Accessing a Value Within an Array

To access a specific value within an array, include its index immediately after the name of the array, in parentheses. The index may also be expressed through a

variable.

```
y = x(4) ' y would get a value of -30, according to the previous array
y = x(z) ' y would get the value of index z within array x.
```

As an example, you could iterate through an array with a loop (such as a [For... Next Statement](#)⁸⁶) and execute code on each element in the array, by using a variable to refer to the index of an element. Let's see how to calculate the sum of the first three elements in the array we previously defined:

```
dim x(5) as char
x(0) = 15
x(1) = 32
x(2) = 4
x(3) = 100
x(4) = -30

dim sum as integer
sum = 0

for i = 0 to 4 ' note that you do not necessarily have to iterate through
all elements in the array.
    sum = sum + x(i)
next i
' now, at the end of this loop, sum contains the sum for the first three
elements (51)
```

The TIDE and Tibbo Basic **V2.0** introduced correct handling of array indices. It is no longer possible for your program to point to an array element that does not exist. For example, if your array only has 5 elements and you try to access element number 5 the Virtual Machine will:

- Generate an [OOR \(Out Of Range\) exception](#)²⁹ and halt if your program is in the [debug](#)²⁷ mode. If you attempt to continue the Virtual Machine will access x(4) -- the element with maximum available index.
- When in the [release](#)²⁷ mode, the OOR exception will not be generated but "index limiting" will still occur.

Example:

```
dim x(5) as char
dim f as byte

f=5
x(f)=3 'index limiting will happen here (preceded by the OOR exception if
you are in the debug mode)
```

Compiler is smart enough to notice out-of-range situations even at compile time. For example, the following code will generate an error during compilation:

```
dim x(5) as char
x(5)=3 'compiler will determine that this operation will be eccessing an
array element which does not exist!
```

Multi-Dimensional Arrays

The array in the example above is called a *one-dimensional* array. This is because every element in the array has just a single index number. However, we could also have an array which looks like this:

Index:	0	1	2	3	4
0	15	32	4	100	-30
1	78	15	-3	0	55
2	32	48	97	5	22
3	13	18	9	87	54
4	32	35	79	124	3
5	7	-9	48	8	99

This is called *two-dimensional array*. Each element in the array is now identified by an index consisting of two numbers (two coordinates). For example, the element **2, 0** contains the value **32**. To create such an array, you would use the following code:

```
dim x(5,6) as char
x(0,0) = 15
x(0,1) = 32
x(0,2) = 4
.....
x(5,2) = 48
x(5,3) = 8
x(5,4) = 99
```

Iterating through such an array would be done using a nested loop for each dimension in the array. The array above contains only two dimensions, so we would nest one loop within another. For an array containing six dimensions, we would have to use six such nested loops. See:

```

dim x(5, 6) as char
x(0, 0) = 15
x(0, 1) = 32
x(0, 2) = 4
.....
x(5, 2) = 48
x(5, 3) = 8
x(5, 4) = 99

dim i, j, sum as integer
sum = 0
for i = 0 to 5
    for j = 0 to 4
        sum = sum + x(i,j)
    next j
next i
' here, sum would be equal to the sum of the whole array. How much is
that? Try and see.

```

In Tibbo Basic, you may define up to 8 dimensions in an array.

Alternative way of defining arrays

We have already explained that the following string means "an array x containing 10 elements of type byte":

```
dim x(10) as byte
```

In Tibbo Basic, the same can be expressed in a different way:

```
dim x as byte(10) 'you can think of it as '10 times of byte type' :-)
```

Both ways of defining arrays are completely identical and you can even mix them together, as we can see on the following examples of 2-dimensional arrays:

```

dim i(20,10) as byte 'two-dimensional array, 20x10 elements
dim i2(20) as byte(10) 'same! that is, 20 groups of byte x 10 -- exactly
same meaning
dim i3 as byte(20,10) 'yet another way -- same result!

```

Now, Tibbo Basic strings can be defined with an optional parameter specifying maximum string length, for example:

```
dim s as byte(30) 'this string will have maximum length of 30 characters
(bytes)
```

So, how do we declare an array of string variables? Here are some examples:

```
dim s(20,10) as string(30) 'two-dimensional array of 30-byte strings,
20x10 elements
dim s2(20) as string(30)(10) 'same!
dim s2 as string(30)(20,10) 'same!
```

Arrays introduce slight overhead

Each array occupies more space than the sum total of space needed by all elements of an array. This is because each array also includes housekeeping data that, for instance, defines how many elements are there in an array, array of what type that is, etc.

Structures 4.6

Beginning with **V2**, Tibbo Basic supports structures. Structure is a combinatorial user-defined data type that includes one or several *member* variables. Structures are declared using **type ... end type**^[94] statements, as shown in the example below:

```
'this is a structure with three members
type my_struct
    x as byte
    y as Long
    s as string
end type
```

In the above example, we declared a structure `my_struct` that has three member variables: `x`, `y`, and `s`. This is just a declaration -- you still have to define a variable with the new type you have created if you want to use the structure of this type in your program:

```
dim var1 as my_struct 'this is how you define a variable with type
'my_struct'
```

After that, you can address individual elements of the structure as follows:

```
var1.x=5
var1.y=12345678
var1.s="Test"
```

Structures you define may include members that are arrays or other structures. Structure variables like `var1` above can be arrays as well, of course. In total, Tibbo

Basic supports up to eight nesting levels (as in "array within a structure within a structure within and array" -- each structure or array is one level and up to 8 levels are possible). Here is a complex example:

```

type foo_struct 'structure with two members, and both are arrays
    x(10) as byte
    s(10) as string(4)
end type

type bar_struct 'structure with two members, one of which is another
structure
    foo as foo_struct 'so, this member is a structure
    w as word
end type

dim bar(20) as bar_struct 'we define an array of type 'bar_struct'

bar(1).foo.s(2)="test" 'oh-ho! we address element 2 of member s of member
foo of element 1 of bar!

```

Structures introduce slight overhead

Each structure occupies more space than the sum total of space needed by all of its members. This is because each structure also includes housekeeping data that, for instance, defines how many members are there, what type they have, etc.

Enumeration Types

At times, it may be useful for a programmer to define his own enumeration data types; for example, when working with the days of the week, it may be useful to refer to them by name in code, rather than by number:

```

dim i as integer
i = 2 ' i is an integer, and can only be assigned a numerical value. you
would have to remember that 2 is Monday.

dim d as dayofweek
d = monday ' now d is a user-defined type, dayofweek, and can be assigned
a value using the symbol Monday.

```

Enumeration definitions are made like this:

```

enum dayofweek
    sunday = 1,      ' if 1 is not specified, the default value associated
    with the first constant is 0.
    monday,          ' by default, increments the previous value by 1. can
    also be explicitly specified.
    tuesday,
    wednesday,
    thursday,
    friday,
    saturday,
    holiday = 99,   ' as described above, values can be explicitly associated
    with any constant in the list.
    holiday2        'this will have the value of 100
end enum

```

An enumeration type would then be used within the code as shown above (d =

Monday). Note that even though Monday is an [identifier](#)^[132] (i.e., an actual word, and not just some number) it does not have to be surrounded by quote marks (because it's not a string).

The value associated with each identifier within the enumeration type doesn't necessarily have to be unique; however, when you associate the same value with several constants, the distinction between these constants will be lost on compile time.

```
enum dayofweek
    Sunday = 1,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    holiday = 99
    bestdayofweek = 7 ' bestdayofweek and Saturday are actually the
same!
    fridaythethirteenth = -666 ' a negative constant.
end enum
```

Note that above, `saturday` was implicitly (automatically) associated with the value 7, and `bestdayofweek` was explicitly associated with the same value. Now, on compile-time, they would both be considered to be just the same.

Type Mapping for Enum Types

When the project is being compiled, all enumeration types are substituted with plain numerical values. the platform on which the code is running doesn't have to know anything about Saturday or about the best day of the week; for the platform, the number 7 is informative enough.

Hence, enumeration types are converted to various built-in numerical variable types. The actual numerical type used to store the enumeration type depends on the values associated with the constants within this enumeration type:

Values associated with constants do not exceed range	Actual variable type used to store enum type
-128 to 127	char
0 to 255	byte
-32768 to 32767	short
0 to 65535	word
-2147483648 to 2147483647	long
0 to 4294967295	dword

Notice, that enumeration types cannot be converted into real values, so you cannot use fractional numbers in your enums.

Some examples:

```

enum tiny
    tinyfoo,
    tinybar
end enum ' this enum will be associated with a char type

enum medium
    mediumfoo = 254
    mediumbar
end enum ' this enum will be associated with a byte type

enum impossible
    baddy = -1
    cruise = 4294967295
end enum ' this enum will raise a compile error, as no single variable
type can hold its values

enumnofractions
    thisisok = 4294967295
    thisisnot = 125.25
end enum ' compiler won't accept this! Integer values only, please!

```

Enumeration types are helpful when debugging your code

For each enumeration type variable, the [watch facility](#)³³ of the TIDE shows this variable's current numerical value with a correct identifier associated with this value. This usually proves to be very useful during debugging! After all, seeing "dayofweek= 2- Monday" is much less cryptic than just "dayofweek= 2"!

Understanding the Scope of Variables

A *scope* is a section of code, from which you can 'see' a variable (i.e, assign it a value, or read its value). For example:

```

sub foobar(x as byte)
    dim i, y as byte
    y = x * 30
    for i = 1 to y
        dim r as short
        r = x + 5
    next i
    r = x + 5 ' this would produce a compiler error
end sub

```

So, in the example above, **x** and **y** could be seen from anywhere within the **sub** procedure called **foobar**. However, **r** could be seen only from within the **for...** **next** statement. Thus, trying to assign **r** a value from outside the **for...** **next** statement would result in a compiler error, because it actually doesn't exist outside of that loop.

One identifier can refer to several different variables, depending on the scope in this identifier is used:

```

dim x as byte ' this creates the variable x in the global scope.

sub foobar(x as byte) ' here we create x once more, in the scope of the
local sub (x as an argument).
    dim f, y as byte
    x = 5 ' right now, only the locally-created x (foobar argument)
equals 5; global x remains unchanged.
    y = x * 30
    for f = 1 to y
        dim x as byte
        x = 30 ' the argument x outside the for... next statement
still equals 5. only this local x equals 30.
    next f
end sub

```

Tibbo Basic supports several scopes:

Global Scope

Every compilation unit has, in itself, one global scope. Variables declared in this scope are accessible from within any **sub** or **function** in this compilation unit.

```

dim s as string

sub foo
    s = "foo" ' assigning a value to the global string variable s.
end sub

sub bar
    dim i as short
    i = 0
    s = "" ' initialize s, in case it contains anything already (such
as 'foo').
    for i = 1 to 5
        s = s + "bar" ' note
    next i
end sub ' at this point, s contains 'barbarbarbarbar'.

```

Local Scope

This is the scope which is between the beginning and the end of each of the following statements:

Beginning	End	Notes
sub	end sub	Cannot be nested.
function	end function	Cannot be nested.
for	next	
while	wend	
if... then... else	end if	No exit statement for if... then... else .
do	loop	

Variables declared in this scope are accessible from within the construct in which they were declared. Local scopes may be nested, for example, **for...next** scope inside **sub...end** sub scope.

A locally defined variable with the same name as a global variable takes precedence in its context over any variable with the same name which is defined in a 'wider' scope.

Local variable names also take precedence over procedure names. For example:

```
sub prc1(x as byte)
    'some code here
end sub

sub prc2
    dim prc1 as byte 'define a local variable with the same name as one
procedure we have
    prc1=0 'this will generate no error -- in the current scope prc1 is a
variable
    prc1(2) 'here, we try to invoke sub prc1 and this will cause a compiler
error.
end sub
```

HTML Scope



This section applies only to platforms which include an HTTP server.

This is a special scope, implemented in Tibbo Basic. HTML files included within a project may contain embedded Tibbo Basic code. This code is executed when the HTTP server processes an HTTP GET (or POST) request. Statements within an HTML file are considered to be within one scope -- similarly to a **function** or **sub** scope, with the exceptions that **include**^[90] and **declare**^[79] statements are allowed.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD W3 HTML//EN">
<HTML>
<BODY>

    BEGINNING OF OUTPUT<br>

    <?
    include "global.tbh"
    declare i as integer ' i is defined somewhere else
    for i = 1 to 10
    ?>

        <i> Foo </i>

    <?
    next i
    ?>

        <br>END OF OUTPUT<br>

    </BODY>
</HTML>
```

Designing dynamic HTML pages always presents you with a choice: what to do-- include the BASIC code right into the HTML file or create a set of subs and functions and just call them from the HTML file? In the first place you put a lot of BASIC code into the HTML file itself, in the second case you just call subs and

functions from the HTML file. So, which way is better?

Generally, we recommend to use the second way. First of all, this style of programming is cleaner- a mixture of BASIC code and static HTML text usually looks messy. Second, the second method *consumes less variable memory*.

Although the HTML scope is similar to a local scope of a function or a sub, its variables get exclusive memory allocation as if they were global. When you avoid writing a lot of BASIC code in the HTML file itself you usually avoid having to create a lot of variables in the HTML scope and this saves you memory!

Declaring Variables

Usually, a variable is first defined by the **dim** statement and then used in code; however, at times, a single global variable must be accessible from several compilation units. In such cases, you must use the **public** modifier when defining this variable, and use the **declare** statement in each compilation unit from which this variable needs to be accessed.

For example, let us say this is the file **foo.tbs**:

```
public dim i as integer ' the integer i is defined in this file, and made
into a public variable. It can now be used from other compilation units.
```

And this is the file **bar.tbs**:

```
i = i + 5 ' this would cause a compiler error. What is i?

' the correct way:
declare i as integer ' lets the compiler know that i is defined elsewhere.
i = i + 5
```

Also, if for some reason you would attempt to use a variable in a single compilation unit before defining it using the **dim** statement, you will have to use a **declare** statement before using it to let the compiler know that it exists. For example:

```
declare i as integer
i = 7 ' i has not been defined yet, but we let the compiler know that it
is defined elsewhere.

...
dim i as integer ' we now define i. Note that here it doesn't have to be
public, because it is used in the same compilation unit.

' this example is rather pointless, but just illustrates this single
principle.
```

Constants.4.10

Constants are used to represent values which do not change throughout the program; these values may be strings or numbers. They may either be stated explicitly, or be derived as the result of an expression.

Some examples:

```

const universal_answer = 42
const copyright = "(c) 2005 Widget Systems Inc." ' this is a string
constant
const escape_char = `@` ' this constant will contain a numerical value --
the ASCII code for the char @.

const hexi = &hFB ' would create a constant with a value of 251 (&hFB in
hex)
const bini = &b00110101 ' would create a constant with a value of 53
(&b00110101 in binary)

const width = 10
const height = 15
const area = width * height ' constants may contain expressions which
include other constants

dim x as byte
const foo = x + 10 ' this will produce a compiler error. Constant
expressions may contain only constants.

```

Constants can be useful when you have some values which are used throughout the code; with constants, you can define them just once and then refer to them by their meaningful name. This has the added benefit of allowing you to easily change the value for the constant any time during the development process -- you will just have to change the definition of the constant, which is a single line of code.



When defining a list of related constants, it is often convenient to use the [Enum Statement](#)⁵⁴ and create one data type which contains this list of constants. See also [User-Defined Types](#)⁵⁵ above.

When defining a constant within a [scope](#)⁵⁷, this constant is visible only from within this scope. It is a good idea to define all constants within header files, and [include](#)⁵⁸ these files into each compilation unit.

String constants

String constants can include escape sequences to define unprintable characters. This functionality is borrowed from C. Adding unprintable characters to the string has always been rather inconvenient in BASIC language. The only way to do so was like this:

```
s = "abc"+chr(10)+chr(13) 'add LF/CR in the end
```

In Tibbo Basic you can achieve the same by using escape sequences -- C style:

```

s = "abc\n\f" ''\n' means LF, '\f' -- CR
const STR1 = "abc\x10\x13" 'same result can be achieved using HEX codes of
the characters

```

The following standard escape sequences are recognized:

- "\0" for ASCII code 0
- "\a" for ASCII code 7 (&h7)
- "\b" for ASCII code 8 (&h8, BS character)

- "\t" for ASCII code 9 (&h9)
- "\n" for ASCII code 10 (&hA, LF character)
- "\v" for ASCII code 11 (&hB)
- "\f" for ASCII code 12 (&hC)
- "\r" for ASCII code 13 (&hD, CR character)
- "\e" for ASCII code 27 (&h1B, ESC character)

Any ASCII character, printable or unprintable, can be defined using its HEX code. The following format should be used: "\x00" where "00" is the HEX code of the character. Notice, that two digits should be present on the code, for example: "\x0A" -- leading zero must not be omitted.

\

Introduction to Procedures

A *procedure* is a named piece of code, which performs a designated task, and can be called (used) by other parts of the program. In Tibbo Basic, there are two types of procedures:

Function Procedures

A function is defined using the [Function Statement](#)⁸⁷. Functions can optionally have one or several arguments. Functions always return a single value. They can, however, change the value of the arguments passed to them using [ByRef](#)⁶⁴ and thus indirectly return more than one value. This would be an example of a function:

```
function multiply(a as integer, b as integer) as integer
    multiply = a * b
end function
```

Note how the function above returns a value: via a local variable with the same name as the function itself. Such a variable is automatically created by the compiler for each function.

Sub Procedures

Sub is short for *subroutine*; just like a function, a sub procedure can optionally accept one or more arguments. However, unlike functions, sub procedures do not return a value. It is defined using the [Sub Statement](#)⁹³. This would be an example of a sub:

```

dim a(10) as byte ' a is a global variable -- outside the scope of the
function.

sub init_array
    dim i as integer
    for i = 0 to 9
        a(i) = 0 ' the global variable gets changed.
    next i
end sub

```

Subs change the value of the arguments passed to them using [ByRef](#) and thus indirectly return a value, or even several values. Of course, they may also change the value of global variables.

Event handlers are like subs

Event handlers defined in the platform work exactly like sub procedures. Event handler subs can accept arguments. Event handlers can never be function procedures as each function has to return a value and the event handler has nobody to return this value to.

Declaring Procedures

Usually, a procedure is first defined by the [function](#) or [sub](#) statements and then used in code; however, at times, functions can reside in a different compilation unit. In such a case, you must use the [public](#) modifier when defining this function, and use the [declare](#) statement to let the compiler know that the function exists.

For example, let us say this is the file **utility_functions.tbs**:

```

public function multiply(a as integer, b as integer) as integer ' the
value returns by this function is an integer
    multiply = a * b
end function

```

And this is the file **program.tbs**:

```

declare function multiply(a as integer, b as integer) as integer '
declaring just the name isn't enough. Include also the arguments and the
types.
dim i as integer
i = multiply(3, 7)

```

Declare statements are usually used within header files which are then included into compilation units. Also, if for some reason you would attempt to use a procedure in a single compilation unit before defining it, you will have to use a **declare** statement to let the compiler know that it exists. For example:

```

declare function multiply(a as integer, b as integer) as integer
dim i as integer
i = multiply(3, 7)

...
function multiply(a as integer, b as integer) as integer ' now this
function doesn't have to be public.
    multiply = a * b
end function

```

Event handler subs require no declaration as they are already declared in your device's platform.

No Recursion

One thing you have to know is that procedures cannot call themselves. Also, two procedures cannot call each other. This is due to TiOS not using dynamic memory allocation. Such allocation would create a serious overhead for the system, and would drastically slow everything down -- not just recursive procedures. For more information, see [Memory Allocation for Procedures](#)⁶⁶.

Passing Arguments to Procedures

When calling subroutines or functions, it is often necessary to pass a certain value for processing within the procedure. An example of this would be a function which calculates the sum of two values; naturally, such a function would need to get two arguments -- the values which are to be added up.

There are two different ways to pass arguments to such a procedure:

The Default: Passing By Value

When passing an argument to a procedure *by value*, this argument is *copied* to a location in memory which was reserved for the local variables of this function. Processing is then done on this local copy -- the original remains untouched. For example:

```

sub foo(x as byte)
...
x = 1
...
end sub

sub bar
    dim y as byte
    y = 2
    foo(y)
    ' at this point in code, y is still 2.
end sub

```

This way of passing variables is the default used in Tibbo Basic.

Passing By Reference

In certain cases, copying is not the preferred solution; for example, when a procedure has to modify several arguments passed to it and these later have to be

accessible. Another example would be when processing large strings -- copying them would cause significant overhead.

In such cases, arguments are passed *by reference*. When passing by reference, the actual values are not copied. Instead, the procedure receives a reference to the location of the original values in memory -- hence, the name. For example:

```
sub foo(byref x as byte)
...
x = 1
...
end sub

sub bar
    dim y as byte
    y = 2
    foo(y)
    ' at this point in code, y is 1!
end sub
```

When passing arguments by reference, the code within the procedure will access these arguments using indirect addressing. This may cause possible overhead. The only case where it does not cause overhead (relative to passing by value) is when working with large strings; in this case, passing them by reference saves the need to copy the whole string to another location in memory.



Here is our advice: when dealing with strings it is usually better (in terms of performance) to pass them by reference. For all other types, passing by value yields better performance.

Strict byref argument match is now required!

Beginning with Tibbo Basic release **2.0**, strict match is required between the type of byref argument and the type of variable being passed. For example, trying to pass a string for a byte will now cause a compiler error:

```
sub bar(byref x as byte)
...
end sub

sub foo
    bar("123") 'attempt to pass a string will generate a compiler error
    bar(val("123")) 'this will work!
end sub
```

In the above example, sub bar takes a byte argument and we are trying to pass a string. Wrong! Compiler understands that byref arguments can be manipulated by the procedure that takes them. Therefore, it is important that actual variables being passed match the type of argument that procedure expects. Automatic [type conversion](#)⁴⁵ won't apply here.

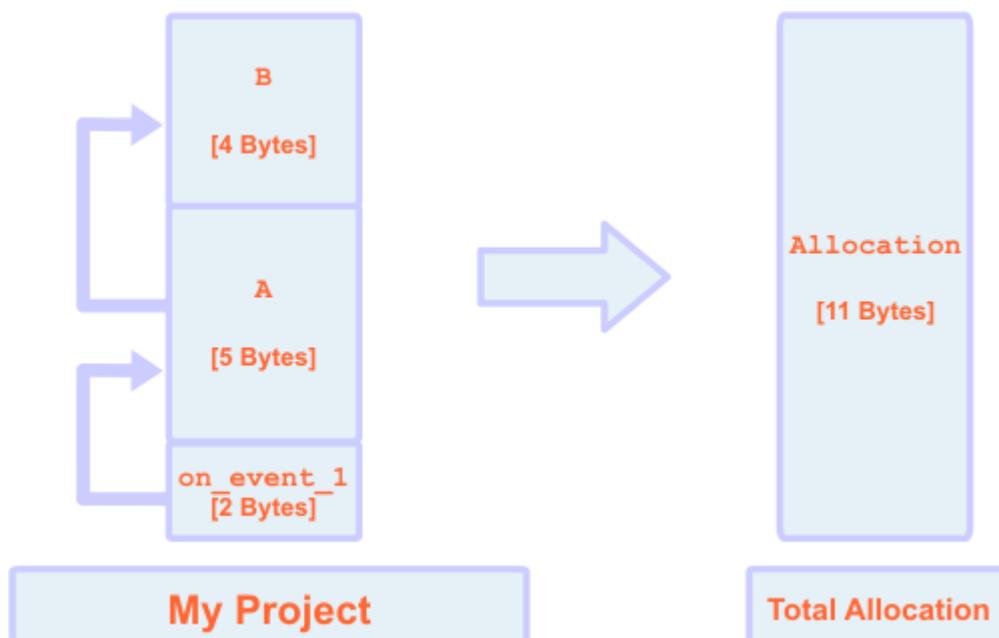
Memory Allocation for Procedures

Variable memory (RAM) allocation in TiOS is [not dynamic](#)^[43]. Memory is allocated for variables at compile-time.

The compiler builds a "tree" reflecting procedure calls within your project. When two different procedures never call each other, it is safe to allocate the same memory space for the variables of each of them. They will never get mixed.

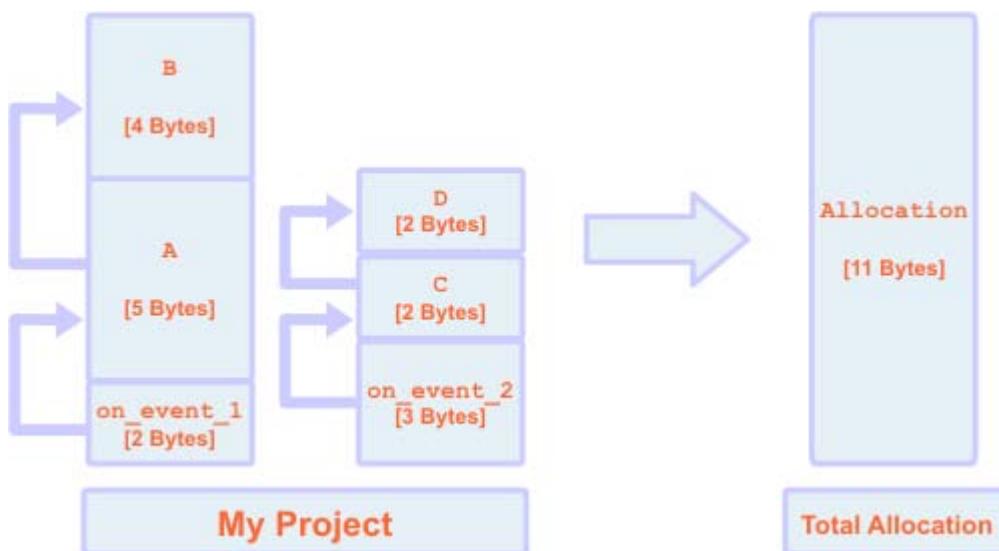
Let us say we have two event handlers in our project: `on_event_1`, which needs 7 bytes of memory, `on_event_2`, which needs 5 bytes of memory. They do not call each other. In this case, the total memory required for our project will be 7 bytes -- they will share the same memory space because only one will be executing at any given time.

However, sometimes procedures call other procedures. This affects memory allocation.



As seen above, the event handler `on_event_1` calls procedure A, which in turn calls procedure B. The memory required for each procedure is listed in brackets. Since `on_event_1`, procedure A and procedure B call each other, they may not share the same memory space. If procedure A keeps a variable in memory, then obviously procedure B cannot use the same space in memory to store its own variables, because procedure A may need its variable once control returns to it after procedure B has completed. Thus, the total memory required for this tree is 11 bytes.

Now, let us say we also have `on_event_2` in our project, which calls procedure C, which in turn calls procedure D. This is a completely separate chain:



As can be seen, this chain takes up 7 bytes of memory. However, this memory can be the same memory used for the `on_event_1` chain, because these two chains will never execute at the same time. Thus, the total memory required for our project remains at 11 bytes.

A typical project usually includes a number of [global variables](#)⁵⁷. Naturally, these variables are allocated their exclusive space that is not shared with local variables of procedures. Variables of HTML scope, which are local by nature, are allocated exclusive memory space as if they were global (this is an unfortunate byproduct of the way compiler handles HTML pages). Hence, it is more economical to implement necessary functionality in procedures invoked from HTML pages rather than include BASIC code directly into the body of HTML files.

Introduction to Control Structures

Control Structures are used to choose what parts of your program to execute, when to execute them, and whether to repeat certain blocks of code or not (and for how many iterations).

The two main types of control structures are [decision structures](#)⁶⁷ and [loop structures](#)⁶⁸.

Decision Structures

Decision Structures are used to conditionally execute code, according to the existence or absence of certain conditions.

Common uses for decision structures are to verify the validity of arguments, to handle errors in execution, to branch to different sections of code, etc.

An example of a simple decision structure would be:

```
dim x, y as byte
dim s as string

if x < y then
    s = "x is less than y"
else
    s = "x is greater than, or equal to, y"
end if
```

The following decision structures are implemented in Tibbo Basic:

- [If.... Then.... Else Statement](#)⁸⁹
- [Select-Case Statement](#)⁹¹

Loop Structures

Loop structures are used to iterate through a certain piece of code more than once. This is useful in many scenarios, such as processing arrays, processing request queues, performing string operations (such as parsing), etc.

An example of a simple loop structure would be:

```
dim f, i as integer
f = 1
for i = 1 to 6
    f = f * i
next i
' f is now equal to 1*2*3*4*5*6 (720).
```

The following loop structures are implemented in Tibbo Basic:

- [Do... Loop Statement](#)⁸²
- [For... Next Statement](#)⁸⁶
- [While-Wend Statement](#)⁹⁵

Doevents^{2.6.3}

Although under Tibbo Basic, event-driven programming is the norm, there may be special cases in which you just have to linger an overly long time in one event handler. This will block execution of other events. They will just keep accumulating in the queue (see [System Components](#)⁷).

To resolve this, a [doevents](#)⁸² statement has been provided. When this statement is invoked within a procedure, the execution of this procedure is interrupted. The VM then handles events which were present in the queue as of the moment of doevents invocation. Once these events are handled, control is returned to the procedure which invoked doevents.

If new events are added to the queue while doevents is executing, they will not be processed on the same doevents 'round'. Doevents only processes those events present in the queue at the moment it was invoked.

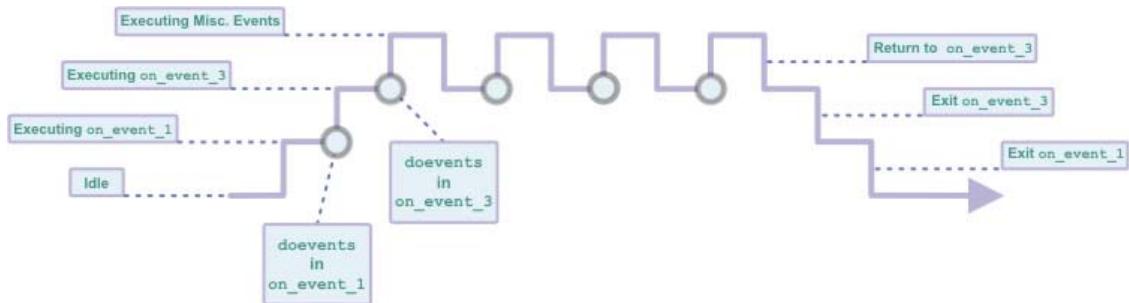
To summarize, doevents provides a way for a procedure to let other events execute while the procedure is doing something lengthy.

```
'calculate sum of all array elements -- this will surely take time!
dim sum, f as word
sum = 0
for f = 0 to 49999
    sum = sum + arr(f)
    doeevents 'don't want to stall other events so allow their execution
while we are crunching numbers
next f
```

Multiple Doevents Calls

Let us say we are in event handler `on_event_1`. This event handler executes a `doevents` call. The VM begins processing other events in the queue, and starts executing an event handler `on_event_3`, which also contains a `doevents` statement.

In this case, a new `doevents` round will begin. Only when it completes, control will be returned to event handler `on_event_3`, which will complete, and then return control to the previous `doevents` (the one from event handler `on_event_1`).

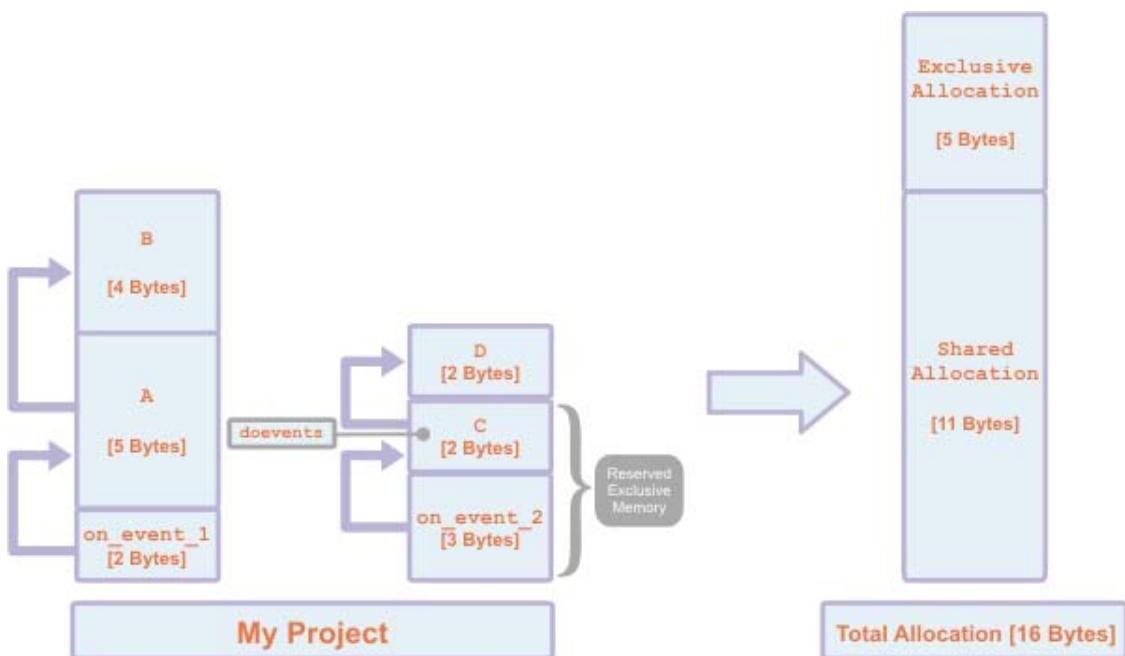


The point here is that control will not be returned to `on_event_1` until `on_event_3` fully completes execution, since `on_event_3` contains a `doevents` statement in itself.

Memory Allocation When Using Doevents

When a procedure utilizes `doevents`, it means its execution gets interrupted, while other procedures get control. We have no way to know which other procedures will get control, as this depends on the events which will wait in the queue.

As a result, a procedure which utilizes `doevents`, and any procedures calling it (directly or through other procedures) cannot share any memory space with any other procedure in the project. This would lead to variable corruption -- procedures newly executed will use the shared memory and corrupt variables which are still needed for the procedures previously interrupted by `doevents`.



Above we have the same two chains of procedures which appear under [Memory Allocation for Procedures](#), with one noticeable difference: Procedure C includes a `doevents` statement. The `on_event_1` chain takes up 11 bytes. But `on_event_2` and procedure C (which together take up 5 bytes) cannot share the same space with the `on_event_1` chain, because when the `doevents` statement is invoked, the state of variables for `on_event_2` and procedure C must be preserved. So these two get their own exclusive memory space.

Procedure D, which is also a part of the `on_event_2` chain, does not get its own exclusive memory space. This is because it comes later on the chain than the procedure which contains the `doevents` statement. There will never be a situation where the variables of procedure D must be preserved while other chains are executing.

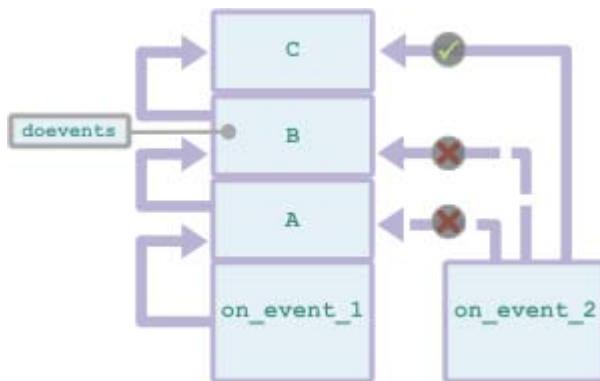
Thus, the total memory requirements of the project depicted above would be 16 bytes -- 11 shared bytes plus 5 exclusive bytes. This is more than would have been required had we not used `doevents`.

Sharing Procedures Which Utilize Doevents

Procedures which contain `doevents` statements, as well as all procedures which call them (directly or through other procedures) cannot be shared between chains.

Let us say event handler `on_event_1` calls procedure A. Procedure A calls procedure B. Procedure B contains a `doevents` statement, and also calls procedure C.

Next, we have event handler `on_event_2`. It cannot call procedure A or B, because procedure B contains a `doevents` statement (and A calls B). If it *could* call procedure A, we might get a situation whereby event handler `on_event_1` fires, calls procedure A. In it, procedure B is called, and `doevents` is executed. During `doevents`, an event handler `on_event_2` fires, and calls procedure A *again* -- while the previous instance of A still holds variables in memory, waiting for control to return to it. This would corrupt the values of variables used by the first A (if you try to do something like this, the compiler will raise an error).



However, note that in the example above we also have procedure C (which is called by procedure B). This procedure can be shared by everyone -- because it is later on the chain than the procedure which contains the doevents statement.

Doevents for Events of The Same Type

For some events, only one instance of the event may be present in the queue at any given time. The next event of the same kind may only be generated after the current one has completed processing. For other events, multiple instances in the queue are allowed.

Let us say that for event_1, multiple instances are allowed, and that this event's handler contains a doevents statement. When this statement executes, it may happen that another instance of event_1 will be found on the queue, waiting to be processed. If this happens, this new instance will just be skipped -- execution will move on to the next event on the queue. Otherwise, we would once again get recursion (execution of an event handler while a previous instance of this event handler is already executing), which is not allowed under Tibbo Basic.

Using Preprocessor

Tibbo Basic compiler includes a preprocessor that understands several directives.

#define and #undef

The **#define** directive assigns a replacement token for an identifier. Before compilation, each occurrence of this identifier will be replaced with the token, for example:

```
#define ABC x(5) 'now ABC will imply 'x(5)'
ABC=20 'now it is the same as writing x(5)=20
```

The **#undef** directive "destroys" the definition made earlier with **#define** :

```
#define ABC x(5) 'define
...
#undef ABC 'destroy
...
ABC=20 'you will get a compilation error on this line (compiler will try
to process this as "=20")
```

#if - #else - #elif -- #endif

These directives are used to conditionally include certain portions of the source code into the compilation process (or exclude from it). Here is an example:

```
#define OPTION 0 'set to 0, 1, or 2 to select different blocks of code for
compilation
...
#ifndef OPTION=0
    s="ABC" 'will be compiled when OPTION=0
#endif OPTION=1
    s="DEF" 'will be compiled when OPTION=1
#ifndef OPTION=2
    s="123" 'will be compiled when OPTION=2, 3, etc.
#endif
```

You can improve on this example and add meaning to 0, 1, and 2:

```
#define RED 0
#define GREEN 1
#define BLUE 2
...
#define OPTION BLUE
...
#ifndef OPTION=RED
    s="ABC" 'will be compiled when OPTION=RED (0)
#endif OPTION=GREEN
    s="DEF" 'will be compiled when OPTION=GREEN (1)
#ifndef OPTION=BLUE
    s="123" 'will be compiled when neither RED, nor GREEN
#endif
```

You can also write like this:

```
#If OPTION
    x=33 'will be compiled in if OPTION evaluates to any value except 0.
    Will not be compiled in if OPTION evaluates to 0.
#endif
```

Preprocessor directives are not "strict". You don't have to define something before using it. During #if evaluation, all undefined identifiers will be replaced with 0:

```
#If WEIRDNESS 'undefined identifier
    x=33 'will not be compiled in
#endif
```

Do not confuse compile-time definitions such as "#define OPTION 2" and actual application code like "const COLOR=2" or "x=3". They are from two different worlds and you can't use the latter as part of #if directives. For example, the following will not work:

```
#define OPTION 0 'preprocessor directive
Const COLOR=2 'constant used by your application

#If OPTION==COLOR 'to a confused programmer, this looks like 0==2, but COLOR
is not #defined, hence, it will be evaluated to 0
  'hence, this code will be compiled in!
#endif
```

The `#if` directive also understands expressions, for example:

```
#define RED 0
#define GREEN 1
#define BLUE 2
...
#define OPTION 2
...
#If OPTION==GREEN+1
  'will be compiled in, because GREEN=1, hence the entire expression
evaluates to 2, and OPTION=2
#endif
```

#ifdef - #else - #endif

`#ifdef` and `#ifndef` are like `#if`, but instead of evaluating an expression they simply checks if specified definition exists:

```
#ifdef OPTION
  s=="X" 'will be compiled if OPTION is defined
#else
  s=="1" 'will be compiled if OPTION is not defined
#endif
```

`#ifndef` is like `#ifdef`, but in reverse:

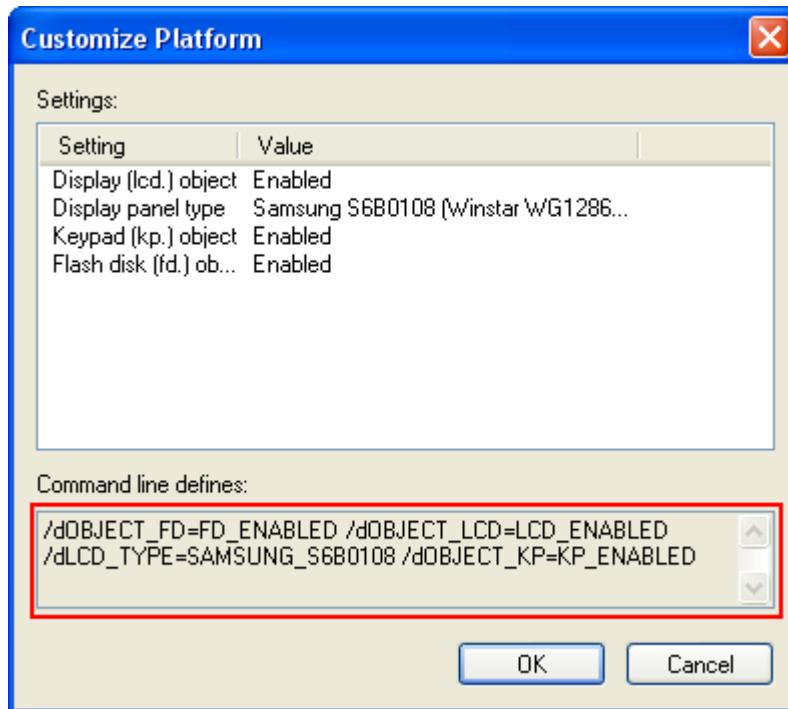
```
#ifndef OPTION
  s=="X" 'will be compiled if OPTION is not defined
#else
  s=="1" 'will be compiled if OPTION is defined
#endif
```

Scope of Preprocessor Directives

Each preprocessor directive applies only to its own [compilation unit](#)³⁹, not the entire project. So, a `#define` directive in *main.h* will not be "visible" in *main.tbs* unless the latter includes the *main.h*.

The only exception to the above are platform defines. These are globally visible throughout your entire project. Platform defines determine options such as the

presence or absence of a display, display type, etc. These options are selected through the Customize Platform dialog, accessible through the [Project Settings](#) dialog.



Working with HTML

One of the strengths of programmable is that they feature a built-in webserver (of course, this is only true for devices that have a network interface and support TCP communications). You can use this webserver as an engine for server-side scripting; simply put, you can output dynamic HTML content by including Tibbo Basic instructions within HTML pages.

Here is a simple example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD W3 HTML//EN">
<HTML>
<BODY>

    BEGINNING OF OUTPUT<br>

    <?
    dim i as integer
    for i = 1 to 10
    ?>

        <i> Foo </i> <br>

    <?
    next i
    ?>

    <br>END OF OUTPUT<br>

</BODY>
</HTML>
```

In effect, this file would cause the following to appear in the browser window of the user accessing this page:

BEGINNING OF OUTPUT

foo
foo
foo
foo
foo
foo
foo
foo
foo
foo

END OF OUTPUT

When to Use HTML Pages In Your Project

For an embedded device, built-in webserver can provide a convenient interface to this device; in fact, it is one of the best ways to allow your users to access your device remotely. They would just have to enter an address in a web browser, and voila, up comes your interface.

HTML support, as implemented in the Tibbo Basic, allows you complete control over page structure. You can use client-side technologies such as JavaScript, CSS etc., while still being able to dynamically generate HTML content within Tibbo device.

Further information about creating HTML files with dynamic content can be found in the [next topic](#)^[76] as well as [Using HTTP](#)^[314] topic (part of the [sock](#)^[274] object documentation).

Embedding Code Within an HTML File

As covered in [Understanding the Scope of Variables](#)⁵⁷, each HTML file has a special scope, and all code within the file resides within this scope.

To begin a block of Tibbo Basic code within an HTML file, you must use an escape sequence -- <? . To close the section of code, use the reverse escape sequence -- ?> .

When the embedded HTTP server receives a GET (or POST) request, it begins to output the requested HTML file. It simply reads the HTML file from top to bottom, and transmits its contents with no alteration. However, the moment it encounters a block of Tibbo Basic code, it begins executing it.

Tibbo Basic code inside HTML files does not differ from the code in "basic" files, but it may not contain procedures. This is because the Tibbo Basic code in the HTML file is considered to constitute a procedure in itself. Notice, that all code in one HTML file is considered to be a single procedure, even if there are several fragments of code in this HTML file. Consider this example:

```
<!DOCTYPE HTML public "-//W3C//DTD W3 HTML//EN">
<HTML>
<BODY>

    BEGINNING OF OUTPUT<br>

<?
'----- BASIC procedure starts here
dim i as integer
for i = 1 to 10
?>

    <i> Foo </i> <br>

<?
next i
'----- procedure ends here
?>

    <br><b>end OF OUTPUT<br>

</BODY>
</HTML>
```

There two code fragments, yet they both form one procedure. For example, variable *i* declared in the first fragment is still visible in the second fragment.

The fact that entire code within each HTML file is considered to be a part of a single procedure has implications in the way events are handled (reminder: there is a [single queue](#)⁷⁴ for all events). The next event waiting in the event queue won't be executed until the end of the HTML procedure is reached. Just because the HTML procedure consists of two or more fragments does not mean that other events will somehow be able to get executed while the HTTP server outputs the static data between those fragments ("<i> Foo </i>
" in our example). Use [doevents](#)⁶⁸ if you want other event handlers to squeeze in!

Tibbo Basic code in the code fragments may include [decision structures](#)⁶⁷ or [loop structures](#)⁶⁸ that may cause various segments of HTML code to be output more than once, to be skipped altogether, or to be output only when certain conditions are true or false. In the above example the line "<i> Foo </i>
" will be output 10 times because this line resides between two code fragments that

implement a cycle!

The same result could be achieved in a different manner:

```
<!DOCTYPE HTML public "-//W3C//DTD W3 HTML//EN">
<HTML>
<BODY>

    BEGINNING OF OUTPUT<br>

    <?
    dim i as integer
    dim s as string

    s=<i> Foo </i> <br>
    for i = 1 to 10
        while sock.txfree<len(s)      'these free lines can be omitted for simple
            doevents                  'tests but are actually essential for
            wend                      'reliable output of large data chunks

            sock.setdata(s)           'this prepares data for output
            sock.send                 'and this commits it for sending (see sock
object)
        next i
    ?>

    <br>end OF OUTPUT<br>

</BODY>
</HTML>
```

Here we have a single code block and "printing" the same line several times is achieved by using [sock.setdata](#)^[353] and [sock.send](#)^[353] methods of the [sock](#)^[274] object.

So, which of the two examples shows a better way of coding? Actually, both ways are correct and equally efficient. The first way will have an advantage in case you have large static blocks that may be harder to deal with when you need to print them using `sock.setdata` method.

Further information about creating HTML files with dynamic content can be found in [Using HTTP](#)^[314] (part of the [sock](#)^[274] object documentation).

Understanding Platforms

As an embedded language, Tibbo Basic may find itself within many various hardware devices; each such device may have different capabilities in terms of storage, physical interfaces, processing power, and other such parameters.

Thus, Tibbo Basic is not a one-size-fits-all affair; it is customized specifically for every type of physical device on which TiOS runs. A function which initializes a WiFi interface would make very little sense on a device which does not support WiFi. The same would go for a function which clears the screen -- what if you have no screen? This holds true even for string functions -- some platforms are so tiny, they do not even need to support string processing!

Because of this, the 'core' of the Tibbo Basic language is actually very minimalistic -- we call it "pure" -- it contains only the statements listed under [Statements](#)^[79] below. Any other functionality is implemented specifically for each platform, and is documented in detail for your platform under [Platforms](#)^[133].

Objects, Events and Platform Functions

Each [platform](#)¹³³ provides the following types of custom language constructs:

Objects

These provide a way to access the various facilities and subsystems of the host platform. For example, in a platform which supports networking, we would have a *socket* object that handles TCP/IP communication.

Each object has *properties*, *methods* and *events*:

A *property* of an object allows you to read or change an internal variable for this object. For example, a *serial port* object may have a *baudrate* property. Change the value of this property, and the actual physical baudrate changes. There are also read-only properties which only provide information.

```
ser.baudrate = 3 ' set the baudrate
x = ser.numofports ' find out how many serial ports the device has.
```

A *method* of an object is a way to make the object perform a certain action. It is basically a procedure. It can optionally take arguments or return values. Our *ser* object could have *getdata* and *setdata* methods, for instance.

```
s = ser.getdata(50) ' gets up to 50 bytes of data into variable s.
ser.setdata(s) ' prepares up to 50 bytes of data for sending.
ser.send ' no arguments, returns nothing. Sends data.
```

An [event](#)⁸ of an object is something that 'happens' to this object in reality. When TiOS registers an event, an *event handler* for it is automatically called, if it exists in your source code. *Event handlers* are simply subs with no arguments.

```
sub on ser_data_arrival
  ' ... do something! ... <-- will be called when data arrives into the
  serial port.
end sub
```

Platform Functions

Many functions commonly available in other BASIC versions are implemented in Tibbo Basic on the platform level and not on the "pure" language level; these include also seemingly universal functions, such as string processing, or various date and time functions. This is done so because not every platform would actually need these functions, universal as they may seem. Some platforms may have very limited resources, and not every platform needs to know what the time is, or how to parse strings.

Language Reference

The text below provides a complete rundown of all built-in language [statements](#)⁷⁹, [keywords](#)⁹⁶ and [operators](#)⁹⁹. If you can't find something here, that means it is platform-specific, and you would find it in your [platform](#)¹³³ documentation.

The examples provided herein may not work on your platform -- they are given for reference only.

Statements

Statements are used for directing the machine to perform a specific operation. A *statement* is the smallest possible unit of code which would compile by itself. You could say they are the programming-language equivalent of a sentence in human speech.

Statements are built into Tibbo Basic itself, and are not platform-specific.

Const Statement

Function:	Declares constants for use in place of literal values.
Syntax:	const name = value
Scope:	Global, HTML and local
See Also:	Enum Statement ⁸⁴

Part	Description
name	The name of the constant, later used to refer to this constant in code.
value	The value for the constant; this can be an expression.

Details

This statement defines an identifier, and binds a constant value to it. During compilation, when the compiler finds an identifier, it substitutes the identifier with the value given for the constant.

When defining a constant, you can use any valid constant expression; this can be another constant, a string, or a mathematical expression. Of course, constant expressions cannot include variables, functions, or other elements of code which are not constant (and, hence, cannot be resolved during compilation).

Global constants are usually declared in the [header file](#)¹⁵.

Examples

```
const foo = "abc" + "fdddf"
const bar = 123 + 56 * 56
const foobar = "dfdffgfg"
```

Declare Statement

Function:	Declares a function or a subroutine or a variable for later use.
------------------	--

Syntax:

declare function name [([**byref**] argument1 **as** type1, [**byref**] argument2 **as** type2,...)] **as** ret_type

or:

declare sub name [([**byref**] argument1 **as** type1, [**byref**] argument2 **as** type2,...)]

or:

declare name [(bounds1)] [, name2 [(bounds2)]] **as** type [(max_string_size)]

Scope:

Global and HTML

See Also:

[Function Statement](#)⁸⁷, [Sub Statement](#)⁹³, [Dim Statement](#)⁸¹

Part	Description
function	Optional. Used to specify that you are declaring a function procedure ⁶² .
sub	Optional. Used to specify that you are declaring a sub procedure ⁶² . If neither sub nor function appear, it is assumed that the declare is used to declare a variable.
name	Required. Used to specify the name of the function, sub or variable you are declaring.
byref	Optional. If present, arguments are passed by reference ⁶⁴ . If not, arguments are passed By Value ⁶⁴ .
argument1[1, 2...]	Optional. The name of the argument to be passed to the procedure. Only used if sub or function appear.
as	Required. Precedes the type definition.
type[1, 2...]	Optional (required if arguments are present). Specifies the type ⁴³ of the argument to be passed to the procedure.
ret_type	Optional (required for functions, cannot be used for subs). Used to specify the type of the return value from the procedure.
bounds[1, 2...]	Optional (used only when declaring variables). Specifies the boundary (finite size) of a dimension in an array.

Details

In large projects, you often define a function or variable in one compilation unit, and use it from other units, so it is external to those units.

The unit which uses this external variable or function should refer to it in a way which lets the compiler know that it does indeed exist externally.

The **declare** statement is used to refer to a variable or function in this manner, but doesn't actually allocate any memory or produce any code; rather, it tells the compiler about this external entity, so that the compiler knows about it and can deal with it (see [Dim Statement](#)⁸¹).

Usually, variables and functions which are shared between compilation units are declared in a header file, and this header is then included in the units (see [Include Statement](#)⁹⁰).

Example

```
declare function hittest(x as integer, y as integer) as boolean
declare sub dosomething (s as byref string)
declare devicestate as integer
```

Dim Statement

Function: Defines a variable and allocates memory for it.

Syntax: [**public**] **dim** name1 [(bounds1)] [, name2 [(bounds2)]]
as type [(max_string_size)]

Scope: Global, HTML and local

See Also: [Declare Statement](#) 

Part	Description
public	Optional; may only be used in a global scope. If present, makes the variable(s) public  .
name[1, 2...]	Required. Specifies the name for the variable.
bounds[1, 2...]	Optional. Specifies the boundary (finite size) of a dimension in an array. Several comma-delimited boundary values make a multi-dimensional array.
as	Required. Precedes the type definition.
type	Required. Specifies the type  of the variable.
max_string_size	Optional (can be used only when <i>type</i> is string). Sets the maximum size for a string (default size is 255 bytes).

Details

The **dim** statement creates a variable in the current scope. It reserves memory space for this variable. Hence, as part of a **dim** statement, you have to specify the **type** of the variable; specifying the type also defines how much memory will be allocated for this variable.

When creating strings, you can explicitly define their maximum size by including it in parentheses immediately following the **string** keyword. The default maximum size of strings is 255 bytes, but if you're sure a string will contain less than 255 bytes of data, it is better to constrain it to a lower size (and thus reduce the memory footprint of your program).

The **dim** statement can also be used to create [arrays](#) . This is done by specifying the number of elements in the array in parentheses immediately following the name of the variable. Multi-dimensional arrays are created by specifying the number of elements in each dimension, separated by commas.

Note that creating a variable using **dim** does not assign any value (i.e, 0) to this variable.

There are alternative ways of specifying the number and size of each array

dimension in an array. Please, examine the examples below.

Examples

```
dim x, y(5) as integer ' x is an integer; y is a one-dimensional array of
5 integers.
dim z(2, 3) as byte ' a two-dimensional array, 2x3 bytes.
dim z2(2) as byte(3) ' same -- a two-dimensional array, 2x3 bytes
dim z2 as byte(2,3) ' same again -- a two-dimensional array, 2x3 bytes

dim s as string(32) ' s is a string which can contain up to 32 bytes
dim s(10) as string(32) 'array of 10 strings with 32-byte capacity
dim s as string(32)(10) 'alternative way to make the same definition
```

Doevents Statement

Function:	Interrupts the current event handler and processes all events in the queue at the moment of invocation.
Syntax:	doevents
Scope:	Global, HTML and local
See Also:	Declare Statement ⁷⁸⁾

Details

Executes events pending in the queue, then returns execution to current event handler. See [doevents](#)⁷⁸⁾ above.

Examples

```
'calculate sum of all array elements -- this will surely take time!
dim sum, f as word
sum = 0
for f = 0 to 49999
    sum = sum + arr(f)
    doevents 'don't want to stall other events so allow their execution
    while we are crunching numbers
next f
```

Do... Loop Statement

Function:	Repeats a block of statements while a condition is True or until a condition becomes True .
------------------	---

Syntax:

```
do [ while | until ] expression
    statement1
    statement2
```

...

[exit do]

...

statementN

loop

or:

do

```
    statement1
    statement2
```

...

[exit do]

...

statementN

loop [**while** | **until**] expression

Scope:

Local and HTML

See Also:

[For... Next Statement](#)⁸⁶, [While-Wend Statement](#)⁹⁵, [Exit Statement](#)⁸⁵

Part	Description
expression	A logical expression which is evaluated either before or after the first time the statements are executed.
statement[1, 2...]	Lines of code to be executed.

Details

The **do-loop** statement repeats a block of code. If the condition (**while** or **until**) is included at the end of the loop (after the **loop** keyword), the block of code is executed at least once; If the condition is included at the beginning of the loop (after the **do** keyword), the condition must evaluate to **true** for the code to execute even once.

Any number of [exit do](#)⁸⁵ statements may be placed anywhere in the **do... loop** as an alternate way to exit the loop. These can be used as an alternate way to exit the loop, such as after evaluating a condition mid-loop using an [if... then](#)⁸⁹ statement. **Exit do** statements used within nested **do-loop** statements will transfer control to the loop which is one nested level above the loop in which the **exit do** occurs.

Examples

```

dim i as integer

' example of the first syntax:
i = 0
do
    i = i + 1
loop until i = 10

' example of the second syntax:
i = 0
do until i = 10
    i = i + 1
loop

```

Enum Statement

Function: Declares a type for an enumeration.

Syntax:

```

enum name
    const1 [ = value1],
    const2 [ = value2],
    ...
end enum

```

Scope: Global and HTML

See Also: [Const Statement](#) 

Part	Description
name	Required. The name of the enum type. The name must be a valid Tibbo Basic identifier, and is specified as the type when declaring variables or parameters of the enum type.
const[1, 2...]	Required. The name for the constant in the enum.
value1	Optional. A value associated with the constant.

Details

Enum types can be useful for debugging purposes. When you add an enum type to a watch, you will see the constant name within the enum, and not a meaningless number.

By default, constants get incremental values, starting from 0 for the first item. You can think of this as a counter, enumerating the items in the list. Explicit values may be specified for any constant on the list. This also sets the counter to this explicit value. The counter will continue to increment from this new value.

Examples

```

enum my_enum
    my_const1, ' implicit value -- 0 is assumed
    my_const2 = 5, ' explicit value of 5, counter is set to 5 too.
    my_const3 ' Counter increments to 6, implicit value of 6.
end enum

```

Exit Statement

Function: Immediately terminates execution of function or a loop.

Syntax:

exit do
exit for
exit function
exit sub
exit while

Scope: Local and HTML

See Also: [Do-Loop Statement](#)^[82], [For... Next Statement](#)^[86], [Function Statement](#)^[87], [Sub Statement](#)^[93], [While-Wend Statement](#)^[95]

Part	Description
exit do	Provides an alternative way to leave a do... loop statement. Can be used only from within such a loop. exit do transfers the control to the statement following the loop statement. When used within nested do... loop statements, exit do transfers control to the loop which is one nested level above the loop in which exit do occurs.
exit for	Provides a way to exit a for loop. Can be used only within for... next or for each... next loops. When using exit for , control is passed to the statement which is immediately after the next statement. When used within nested for loops, exit for transfers control to the loop which is one nested level above the loop in which exit for occurs.
exit function	Exits the current function. Execution resumes from the point where the function was originally called.
exit sub	The same as exit function , only used for subroutine procedures.
exit while	Exists a while loop before the condition it is dependant upon evaluates as false .

Details

Do not confuse **exit** (which just quits) with **end** (which defines the end of a section of code).

Examples

```
function send_data as integer
    ' before sending data, we want to make sure Ethernet interface is OK
    if net.failure <> 0 then
        send_data = 1 ' this way we notify the caller of an error
        exit function
    end if
    ' Ethernet interface is OK, proceed with sending data
end function
```

For... Next Statement

Function:	Repeats a block of statements while a counter increments until it reaches a set value.
Syntax:	<pre>for name = start_expression to end_expression [step step_number] statement1 statement2 ... [exit for] ... statementN next name</pre>
Scope:	Local and HTML
See Also:	Do-Loop Statement ⁸² , Exit Statement ⁸⁵ , While-Wend Statement ⁹⁵

Part	Description
name	Required. The name of the counter. This has to be a numeric variable, which was previously explicitly defined using a Dim Statement ⁸¹ .
start_expression	Required. The initial value of the counter. This actually sets the value of the <i>name</i> counter to the result of this expression. Can be a numerical constant, or a more complex expression.
end_expression	Required. The end value for the counter; once the counter reaches it, execution of the for... next loop stops. Can be a numerical constant, or a more complex expression.
step_number	Optional. Defines the intervals in which the <i>name</i> counter is incremented on every pass of the loop. This must be a numerical constant, and can be either positive or negative.
statement[1, 2...]	Required. Lines of code to be executed as long as the counter is 'in range' -- between the <i>start_expression</i> and the <i>end_expression</i> .

Details

It is not advised to change the value of the counter while within a **for... next** loop. You might get unexpected results, such as infinite loops.

For... next loops may be nested, using different counter variables (*name* above). There is also an [exit](#)⁸⁵ statement which can be used to terminate them abruptly.

Under Tibbo Basic, you are required to explicitly state the name of the cycle variable to be incremented immediately following the **next** keyword.

Examples

```

dim a(10) as integer
dim f as byte

for f = 0 to 9 ' in a 10-member array, element indices are 0 to 9
    sum = sum + a(f)
next f

```

Function Statement

Function:	Used to define functions ⁶² -- distinct units in code which perform specific tasks and always return a value.
Syntax:	<pre> [public] function name [([byref] argument1 as type1, [byref] argument2 as type2...)] as ret_type statement1 statement2 ... [exit function] ... statementN end function </pre>
Scope:	Global
See Also:	Declare Statement ⁷⁹ , Exit Statement ⁸⁵ , Sub Statement ⁹³

Part	Description
public	Optional. If set, allows other compilation units (files in a project) to access the function.
name	Required. Specifies the name for the function (used to call it, etc).
byref	Optional. If present, the argument immediately following this modifier will be passed by reference ⁶⁴ .
argument[1, 2...]	Optional. The name of the argument(s) passed to the function; arguments must have a name which is a valid identifier. This is a local identifier, used to refer to these arguments within the body of the function.
as	Optional (required if arguments are specified). Precedes the type definition.
type[1, 2...]	Optional (required if arguments are specified). Specifies the data type ⁴³ for the argument. Each argument name must be followed with a type definition, even when specifying several arguments of the same type.
ret_type	Required. Specifies the type of the value the function will return. In effect, this is the data type of the function.
statement[1, 2...]	Required. The body of code executed within the function; specifies the actual 'work' done by the function.

Details

Functions cannot be nested (which is why their scope is defined as global or HTML). Function always return a single value. Functions can call other functions and subroutines.

The return value of a function must be explicitly set from within the body of the function, by referring to the name of the function as a variable (of type *ret_type*) which is then assigned a value.

Examples

```
'this is just an example to show how functions call each other. It's not
actually useful.
declare subtract (x as byte, y as byte) as integer ' have to declare,
because it's invoked before its body.

function distance(x as byte, y as byte) as integer
    if x>y then
        distance = subtract(x,y)
    else
        distance = subtract(y,x)
    end if
end function
...
function subtract (x as byte, y as byte) as integer
    subtract = x - y
end function
```

Goto Statement

Function: Jumps to a specific point in code, marked by a label.

Syntax: **goto** label

...
label:

Scope: Local and HTML

See Also: ---

Part	Description
label	Required. Marks a specific point in code.

Details

Unconditionally jumps to label in code. Notice that all goto labels are local -- you cannot use goto statement to jump from within one procedure into another procedure!

Examples

```

dim arr1(5),arr2(5),f as byte

sub on_sys_init

arr1(0) = 1
arr1(1) = 2
arr1(2) = 3
arr1(3) = 4
arr1(4) = 5

arr2(0) = 1
arr2(1) = 2
arr2(2) = 2
arr2(3) = 4
arr2(4) = 5

'compare arrays and jump if not exactly the same
for f=0 to 4
    if arr1(f)<>arr2(f) then goto not_the_same
next f
'here when both arrays contain the same data
exit sub

'here when arrays are not the same
not_the_same:
'... place code here ...

```

If.... Then... Else Statement

Function: A way to conditionally execute code.

Syntax:

```

if expression then
    statement1
    statement2
    ...
    [else
        statement1
        statement2
        ...
    ]
end if

```

or:

```
if expression then true_statement1 : true_statement2 ...
```

Scope: Local and HTML

See Also: [Select-Case Statement](#) 

Part	Description
expression	Required. The result of this expression (true or false) is then used to determine what code will execute.
statement[1, 2...]	Required. Statements to execute.

: Optional. Separator for multiple statements on a single line.
Covered under [Programming Fundamentals](#)³⁹.

Details

When the expression evaluates to **true**, the block of code immediately following the **then** keyword is executed. If it evaluates to **false**, the code immediately following the **else** keyword is executed; if there is no **else** keyword, program flow resumes from the line immediately following the **end if** keyword.

When using the single-line syntax (as in the lower example above), an **if** statement must not be terminated using an **end if**. This is the only construct under Tibbo Basic where line end matters. So, if you want to have several statements in such a construct, you need to place them all in the same line, and separate them with colons.

If... then... else statements may be nested.



Currently, single-line **if... then** statements cannot contain an **else** clause.

The **elseif** syntax is not currently supported at all (even on multi-line **if... then** statements).

Examples

```
if net.failure=1 then
    if sys.runmode=0 then ser.setdata("Ethernet failure!"):ser.send
'massage when in debug mode
    sys.halt
else
    if net.linkstate=0 then
        ser.setdata("No Ethernet link!")
    else
        if net.linkstate=1 then
            ser.setdata("Linked at 10Mbit/s.")
        else
            ser.setdata("Linked at 100Mbit/s.")
        end if
    end if
    ser.send
end if
```

Include Statement

Function:	Includes a file (such as a header file) at the point of the statement.
Syntax:	include "filename"
Scope:	Global and HTML
See Also:	---

Part	Description
------	-------------

"filename" Contains the filename to be included. The path can be a relative path to the [project path](#)¹⁵, an absolute path (such as c:\myfolder\myfile.tbs) or even a UNC path (such as \\MY-SERVER\Main\myfile.tbs).

Details

Makes compiler include the contents of a file at the point of the include statement. Usually used to include header files with [declarations](#)⁷⁹, definitions for [constants](#)⁷⁹, [enum types](#)⁸⁴, etc.

Examples

File: global.tbh (a header file)

```
Declare Function multiply(x As Byte, y As Byte) As Integer
Const k=3 'a crucially vital global constant.
```

File: main.tbs (a source code file)

```
Include "global.tbh" ' now we have access to multiply and to K.

Sub on_sys_init
    ...
    Dim result As Integer
    result = multiply(k, 3)
    ...
End Sub
```

File: library.tbs (a source code file)

```
Include "global.tbh"

Public Function multiply(x As Byte, y As Byte) As Integer
    multiply = x * y
End Function
```

Select-Case Statement

Function: A way to conditionally execute code.

Syntax:

```

select select_expression
  case expression1_1 [ , expression1_2, ... ] [ : ]
    statement1_1
    statement1_2
    ...
  case expression2_1 [ , expression2_2, ... ] [ : ]
    statement2_1
    statement2_2
    ...
  ...
  [ case else [ : ]
    statementN_1
    statementN_2
    ...
  ]
end select

```

Scope:

Local and HTML

See Also:[If.... Then... Else Statement](#)

Part	Description
select_expression	Required. The expression which is evaluated first; subsequent expressions are tested to match this expression. If a match is found, the statements contained within this case clause are executed, and execution then resumes from the line immediately following end select .
expression[1_1... N_1]	Required. An expression to evaluate; If it matches the select_expression, the statements included in this case clause are executed.
statement[1_1... N_1]	Required. Statements to execute when expression[1_1... N_1] matches select_expression.
:	Optional. Maintained for backwards compatibility -- some versions of BASIC in the past required a colon following every case expression.
case else	Optional. Precedes a block statements which is executed if neither of the earlier case clause match the select_expression. If present, must be the last case clause.

Details

It is of note that once a matching **case** clause is found, no other **case** clauses are tested; the code within the matching clause is simply executed, and execution resumes from the line following **end select**.

Also, remember that writing two select_expressions at the same time does not mean that there will be a shared code for both of them. Rather, it means that the first expression will have no code associated with it!

```

...
case 1 : 'this expression has no code associated with it
case 2 : 'code 'x=5' belongs to this expression
    x=5
...

```

Correct way to have a single block of code for two expressions is as follows:

```

...
case 1,2 : 'x=5 will be done both for 1 and 2
    x=5
...

```

Examples

```

sub print_weekday (weekday as byte)
    select case weekday
        case 1 : ser.setdata("Monday")
        case 2 : ser.setdata("Tuesday")
        case 3 : ser.setdata("Wednesday")
        case 4 : ser.setdata("Thursday")
        case 5 : ser.setdata("Friday")
        case 6 : ser.setdata("Saturday")
        case 7 : ser.setdata("Sunday")
        case else : ser.setdata("Did you just invent a new day?")
    end select
    ser.send
end sub

```

Sub Statement

Function: Used to define [subs](#) -- distinct units in code which perform specific tasks. These never return any value.

Syntax:

```

[ public ]
sub name [ ( [ byref ] argument1 as type1, [ byref ]
argument2 as type2... ) ]
    statement1
    statement2
    ...
    [exit sub]
    ...
    statementN
end sub

```

Scope: Global

See Also: [Declare Statement](#)⁷⁹, [Exit Statement](#)⁸⁵, [Function Statement](#)⁸⁷

Part	Description
------	-------------

public	Optional. If set, allows other compilation units (files in a project) to access the function.
name	Required. Specifies the name for the function (used to call it, etc).
byref	Optional. If present, the argument immediately following this modifier will be passed by reference ⁶⁴ .
argument[1, 2...]	Optional. The name of the argument(s) passed to the function; arguments must have a name which is a valid identifier. This is a local identifier, used to refer to these arguments within the body of the function.
as	Optional (required if arguments are specified). Precedes the type definition.
type[1, 2...]	Optional (required if arguments are specified). Specifies the data type ⁴³ for the argument. Each argument name must be followed with a type definition, even when specifying several arguments of the same type.
statement[1, 2...]	Required. The body of code executed within the function; specifies the actual 'work' done by the function.

Details

Subroutines cannot be nested (which is why their scope is defined as global or HTML). Subroutines do not return values. Subroutines can call other subroutines and functions.

Examples

```
sub print_to_serial(s as byref string)
    ser.setdata(s)
    ser.send
    s = "OK" ' This sub actually returns something, if indirectly.
end sub
```

Type Statement

Function: Used to declare [structures](#)⁵⁴ -- combinatorial data types that includes one or several *member* variables.

Syntax:

```
type type_name
    name1 [ (bounds1) ] as type [ (max_string_size) ]
    ...
    nameN [ (boundsN) ] as type [ (max_string_size) ]
end type
```

Scope: Global, HTML and local

See Also: [Dim Statement](#)⁸¹

Part	Description
------	-------------

type_name	Required. Specifies the name for this structure type (not a particular variable).
name[1, 2...]	Required. Specifies the name for the member variable.
bounds[1, 2...]	Optional. Specifies the boundary (finite size) of a dimension in an array. Several comma-delimited boundary values make a multi-dimensional array.
as	Required. Precedes the type definition.
type	Required. Specifies the type ⁴³ of the variable.
max_string_size	Optional (can be used only when type is string). Sets the maximum size for a string (default size is 255 bytes).
end type	Required. Closes type declaration.

Details

Structures can include any number of members, and each member can be of any type. Any member can also be an array or another structure. Nesting of up to 8 levels is allowed (i.e. "array within a structure within a structure within an array" -- each structure or array is one level and up to 8 levels are possible).

Type...end type is only a declaration, not a variable definition! You still need to use a regular [dim](#)⁸¹ statement to define a variable of the type you have declared.

Examples

```
'declare new type
type my_struct
    x as byte
    y as Long
    s as string(10)
end type

'define a variable of this type
dim my as my_struct
```

While-Wend Statement

Function: Executes a block of code as long as an expression evaluates to **true**.

Syntax:

```
while expression
    statement1
    statement2
    ...
    [exit sub]
    ...
    statementN
wend
```

Scope: Local and HTML

See Also: [Do-Loop Statement](#)⁸², [For... Next Statement](#)⁸⁶, [Exit Statement](#)⁸⁵

Part	Description
expression	Required. The expression which is to be evaluated.
statement[1, 2...]	Required. The code to run when the expression is true .

Details

Makes a pre-conditional loop. First, the expression is evaluated and then if it is **true** statement1, statement2, etc are executed. Then expression is evaluated again and so on until expression becomes **false**.

Examples

```
function wait_char(ch as byte) as byte
    ' waits for specific character with ASCII code ch to arrive into the
    ' serial port.
    ' returns 0 if char was encountered or 1 if this character was encountered

    dim s as string(1)

        ' input data byte by byte for as long as there is some data left to
        process
        s = ser.getdata(255) ' will input byte by byte as s only can contain
        a single char!
        while len(s)<>0
            if s = chr(ch) then
                wait_char = 0 ' character encountered!
                exit function
            end if
            s = ser.getdata(255)
        wend

        wait_char = 1 ' did not encounter ch character and there is no more
        data to input (for now)!

End Function
```

Keywords

This chapter contains links from single keywords to the statements in which you may find them. It is meant to be used as a resource for context-sensitive help.

As

A keyword designating data type. Appears as part of the following statements:

[Declare Statement](#) 

[Dim Statement](#) 

[Function Statement](#) 

[Sub Statement](#) 

Boolean

A data type. Please see [Variables And Their Types](#)⁴³.

ByRef

A keyword designating a way of passing arguments. Appears as part of the following statements:

[Function Statement](#)⁸⁷

[Sub Statement](#)⁹³

For further details, see [Passing Arguments to Procedures](#)⁶⁴.

Byte

A data type. Please see [Variables And Their Types](#)⁴³.

ByVal

A keyword designating a way of passing arguments. Appears as part of the following statements:

[Function Statement](#)⁸⁷

[Sub Statement](#)⁹³

For further details, see [Passing Arguments to Procedures](#)⁶⁴.

Char

A data type. Please see [Variables And Their Types](#)⁴³.

Else

A keyword used to denote conditions. Appears as part of the following statements:

[If.... Then... Else Statement](#)⁸⁹

[Select-Case Statement](#)⁹¹

End

This keyword is used in the following statements:

[Enum Statement](#)⁸⁴

[Function Statement](#)⁸⁷

[Select Statement](#)⁹¹

[Sub Statement](#)⁹³

[Type Statement](#)⁹⁴

False

A byte constant with a value of 0, associated with [boolean](#)⁴³ variables.

For

Appears as part of the following statement:

[For... Next Statement](#)⁸⁶

Integer

A data type. Please see [Variables And Their Types](#)⁴³.

Next

Appears as part of the following statement:

[For... Next Statement](#)⁸⁶

Public

A keyword used to denote visibility. Appears as part of the following statements:

[Dim Statement](#)⁸¹

[Function Statement](#)⁸⁷

[Sub Statement](#)⁹³

For further details, see [Understanding the Scope of Variables](#)⁵⁷.

Short

A data type. Please see [Variables And Their Types](#)⁴³.

Step

Appears as part of the following statement:

[For... Next Statement](#)⁸⁶

String

A data type. Please see [Variables And Their Types](#)⁴³.

Then

A keyword used to denote conditional execution. Appears as part of the following statement:

[If.... Then... Else Statement](#)⁸⁹

Type

Appears as part of the following statement:

[Type Statement](#)⁹⁴

To

Appears as part of the following statement:

[For... Next Statement](#)⁸⁶

True

A byte constant with a value of 1, associated with [boolean](#)⁴³ variables.

Word

A data type. Please see [Variables And Their Types](#)⁴³.

Operators

Tibbo Basic supports the following operators:

+ Operator

Addition operator (applies to strings as well).

```
i = 1 + 2 ' this would be 3
s = "foo" + "bar" ' this would be "foobar"
```

* Operator

Multiplication operator.

```
i = 5 * 2 ' 10.
```

- Operator

Subtraction Operator.

```
i = 20 - 5 ' 15
```

/ Operator

Division operator.

```
i = 30 / 10 ' 3
i = 10 / 3 ' also 3 -- only integers are supported, decimal part is
removed.
```

MOD Operator

Used to divide two numbers and return the remainder.

```
i = 10 mod 3 ' this would be 1
```

= Operator

(1) Equality operator. **(2)** Assignment operator.

```
if i = 5 then .... ' as an equality operator
i = 5 ' as an assignment operator
```

AND Operator

(1) Logical AND. **(2)** Bitwise AND.

```
if i = 5 AND j = 10 then.... ' as a logical AND

x =
&b01011001 AND
&b10101011 ' this would be
&b00001001
```

NOT Operator

(1) Logical NOT. **(2)** Bitwise NOT.

```
if NOT b then.... ' as a logical NOT -- b is a boolean value
x = NOT &b01011001 ' this would be &b10100110
```

OR Operator

(1) Logical OR. **(2)** Bitwise OR.

```
if i = 5 OR j = 10 then.... ' as a logical OR

x =
&b10110101 OR
&b01011001 ' this would be
&b11111101
```

XOR Operator

(1) Logical XOR. (2) Bitwise XOR.

```
if i = 5 XOR j = 10 then.... ' as a logical XOR  
&b10110101 XOR  
&b01011001 ' this would be  
&b11101100
```

Error Messages

Below is a listing of all error messages which may appear when trying to build and upload your Tibbo BASIC program onto a target.

C1001

Description:

This error occurs when a source file contains an “illegal” character (like a special character, a non-English letter, etc) outside of a string literal or a comment.

Example:

```
%dim x as byte ' error C1001: invalid char '%' (25)
```

See Also

- [Naming Conventions](#) 

C1002

Description:

This error occurs when source code contains a line break inside a string literal or a character constant.

Example:

```
S = "I am a string  
literal " ' error C1002: newline in constant
```

See Also

- [Variables And Their Types](#) 

C1003

Description:

Unlike string literals, character constants may not be empty.

Example:

```
x = `` ' error C1003: empty char constant
```

See Also

- [Constants](#) 

C1004

Description:

Character constants may contain two characters at most (in which case, the constant type will be 'word').

Example:

```
x = `abc` ' error C1004: too many chars in constant
```

See Also

- [Constants](#) 

C1005

Description:

The compiler detected an attempt to specify a numerical constant using an incorrect format.

Example:

```
x = &G12 ' error C1005: invalid numeric constant: unknown base 'G'
x = &b102 ' error C1005: invalid numeric constant
x = 10a ' error C1005: invalid numeric constant
```

See Also

- [Type Conversion](#) 

C1006

Description:

This error occurs when Tibbo BASIC syntax is violated. Refer to the documentation of a particular statement to see its syntax.

Example:

```
select x ' error C1006: 'Case' expected
case 1: case 2:
    b = true
case 3:
    b = false
case else:
    sys.halt
end select

sub on_init
...
end function ' error C1006: 'End Sub' expected
```

See Also

- [Language Reference](#) ↗⁷⁸

C1007

Description:

A numerical constant does not fit in any supported numerical type.

Example:

```
x = 65536 ' error C1007: Constant too big
x = -32769 ' error C1007: Constant too big

enum my_enum
    my_val1 = -32768,
    my_val2 = 32768
end enum ' error C1007: enum range is too wide
```

See Also

- [Variables And Their Types](#) ↗⁴³
- [Constants](#) ↗⁶⁰

C1008

Description:

This error occurs when an expression which should be constant is not actually constant.

Example:

```
dim x as integer
const a = 5 * x ' error C1008: constant expression expected
```

See Also

- [Constants](#)⁶⁰

C1009

Description:

This error occurs when the requested operation cannot be performed using the data types provided.

Example:

```
dim s as string
s = "abc" + 5 ' error C1009: type mismatch
```

See Also

- [Variables And Their Types](#)⁴³

C1010

Description:

This error occurs when attempting to re-use an identifier that is already used (and thus, cannot be reused in the current scope) in a definition or a declaration (of an enumeration type, a constant, a variable or a function).

Example:

```
dim x as string
sub x ' error C1010: redefinition of identifier 'x'
...
end sub
```

See Also

- [Identifier](#) [132]

C1011

Description:

This error occurs when attempting to define a procedure twice.

Example:

```
sub x
end sub

sub x  ' error C1011: 'x' already has body
end sub
```

See Also

- [Sub Statement](#) [93]
- [Function Statement](#) [87]

C1012

Description:

The definition for a procedure does not match a previous declaration for that procedure (a different number or type of arguments and/or a return value)

Example:

```
declare sub x

sub x(i as integer) ' error C1012: argument count mismatch (see previous
declaration of 'x')
end sub
```

See Also

- [Declare Statement](#) [79]
- [Sub Statement](#) [93]
- [Function Statement](#) [87]

C1013

Description:

This error occurs when a statement references an identifier which has not previously been defined.

Example:

```
dim x as integer
x = 15 * y ' error C1013: undeclared identifier 'y'
```

See Also

- [Dim Statement](#)⁸¹

C1014

Description:

Identifiers can refer to entities of different types: labels, variables, procedures, enumeration types, constants. This error occurs when you cannot use identifier of a certain type in the current statement.

Example:

```
dim x as integer
goto x ' error C1014: 'x' is not a label
```

See Also

- [Identifier](#)¹³²

C1015

Description:

The **next** clause of a **for/next** statement must use the same index variable as the **for** clause.

Example:

```
dim i,j as integer
for i = 1 to 10
next j ' error C1015: 'For'/'Next' arguments mismatch
```

See Also

- [For... Next Statement](#) [86]

C1016

Description:

The **exit** statement may be used only from within certain statements (**for**, **while**, **do-loop**, **sub**, **function**). This error occurs when the compiler encounters an **exit** statement which is used not from within one of these statements.

Example:

```
for i = 1 to 10
    exit while  ' error C1016: 'Exit' is of scope
next i
```

See Also

- [Exit Statement](#) [85]

C1017

Description:

Assignment statements must contain a value on left side of the equal sign (called an l-value). This can be a variable, an array element or a read-write property, and may not be a constant.

Example:

```
1 = x  ' error C1017: l-value expected on the left of '='
```

C1018

Description:

Assignment statements must contain a value on right side of the equal sign (called an r-value). This error occurs when an expression on the right side of such a statement does not return a value.

Example:

```
sub sub1
...
end sub
...
x = sub1 ' error C1018: subroutine cannot be on the right of '='
```

C1019

Description:

This error occurs when attempting to access a variable which is not an array as if it were an array (by using an index number).

Example:

```
dim i,j as integer
i = j(3) ' error C1019: j is not array
```

See Also

- [Introduction to Variables, Constants and Scopes](#) 43
- [Arrays](#) 50

C1020

Description:

This error occurs when trying to define an array with more than 8 dimensions (a maximum of 8 dimensions are allowed).

Example:

```
dim i(2,2,2,2,2,2,2,2) as integer ' error C1020: Too many array
dimensions (8 max)
```

See Also

- [Arrays](#) 50

C1021

Description:

This error occurs when a property was defined for read-only or for write-only, but

the program tries to access this property in a different way.

Example:

```
sys.runmode = PL_SYS_MODE_DEBUG  ' error C1021: write access to property  
is denied
```

See Also

- [Understanding Platforms](#) 

C1022

Description:

There are several system calls which the compiler uses directly, and are invoked implicitly in code (string comparison, string copy, conversion from string to number, etc). This error occurs when a platform does not export these functions, but source code requires them.

Example:

```
doevents ' error C1022: platform does not export 'doevents' syscall
```

See Also

- [Type Conversion](#) 

C1023

Description:

The global scope may contain only declarations/definitions of procedures, variables, enumeration types and constants. This error occurs when some other statement is encountered at the global scope.

Example:

```
for i = 1 to 10  ' error C1023: Unexpected at global scope  
next i
```

See Also

- [Understanding the Scope of Variables](#) 

C1024

Description:

This error occurs when an **include** statement references a file which cannot be read. Most commonly, it means the filename was misspelled.

Example:

```
include "utileties.tbh"  ' error C1024: unable to read file
'utileties.tbh'
```

See Also

- [Include Statement](#)

L1001

Description:

This error occurs when the linker tries to link two object files which have different data base addresses.

Lower addresses are reserved for passing arguments and returning values from platform syscalls. The data base address for program variables is calculated according to how much memory platform syscalls require for arguments and return values.

Most commonly, this error means that you are trying to link object files built for different platforms.

See Also

- [System Components](#)

L1002

Description:

This error occurs when the linker attempts to link two object files with different counts of platform event handlers. Most commonly this error means that you are trying to link object files built for two different platforms.

See Also

- [System Components](#)

L1003

Description:

During linking, one or more addresses remained unresolved. That means that these addresses are referenced from one or more compilation units but are never defined.

See Also

- [Declare Statement](#) 

L1004

Description:

This error means that when linking two object files, the linker encountered a situation where the import is a data address and the export is a code address or vice versa.

L1005

Description:

Since all memory allocation is static, recursion is not supported. This error occurs whenever the linker encounters recursion (direct or indirect).

See Also

- [Introduction to Procedures](#) 
- [Our Language Philosophy](#) 

L1006

Description:

The TiOS Virtual Machine may hold up to 255 stack locations. This error occurs when the linker needs to reserve more stack locations. That does not necessarily mean that the direct call chain in program is 255 calls long. Each independent **doevents** statement approximately doubles the required needed stack locations.

See Also

- [Doevents](#)⁶⁸
- [System Components](#)⁷

L1007**Description:**

A function which contains a **doevents** statement may not be called from more than one independent call chain.

See Also

- [Doevents](#)⁶⁸

L1008**Description:**

The amount of RAM needed to store variables for your project exceeds the maximum possible size for this platform. In simple terms, your variables take up too much space.

See Also

- [Platform Specifications](#)¹³³

L1009**Description:**

The amount of FLASH (program memory) needed to store your project exceeds the maximum possible size for this platform. I.e, your program is too large.

See Also

- [Platform Specifications](#)¹³³

Objects, Properties, Methods, Events

Under Tibbo Basic, these are all platform-specific constructs.

Please refer to your [Platform](#)^[133] documentation for details about the objects, properties, methods and events for your platform.

Development Environment

Below is an overview of the TIDE GUI.

As a general rule, those things which you know from common Windows programs (such as Window > Tile) will work just as you would expect them to work. We will concentrate on the more unique parts of the TIDE GUI.

Installation Requirements

The recommended system requirements for TIDE are:

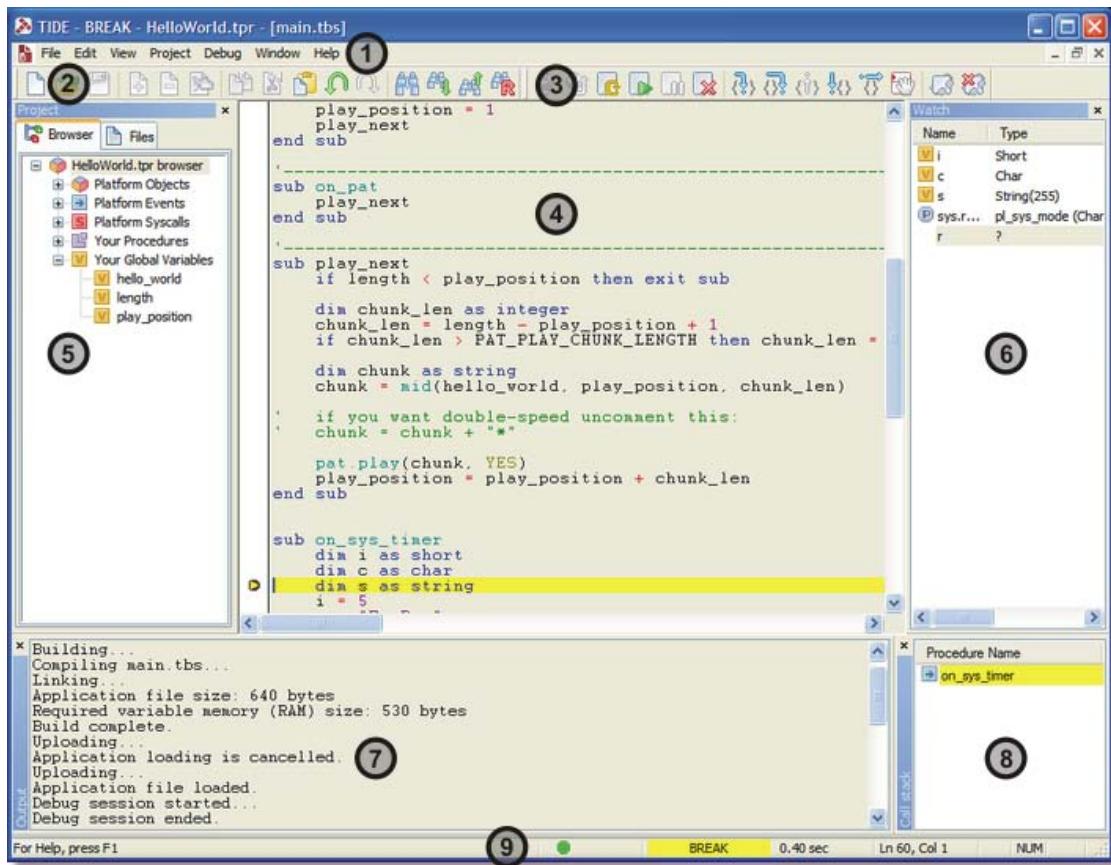
- 800 MHz Intel Pentium III processor (or equivalent) and later
- Windows 98, ME, 2000, XP, 2003
- 256 MB RAM
- 20MB of available disk space
- A communication medium with your target (platform-specific)
- At least one target device to work with (required for debugging)
- 1024 x 768, 24-bit display recommended

User Interface

The following is a systematic overview of the TIDE graphical user interface.

Main Window

The main window for TIDE looks like this:



The parts in the screenshot above are:

- (1) The [Menu Bar](#)
- (2) The [Project Toolbar](#)
- (3) The [Debug Toolbar](#)
- (4) The [Code Editor](#)
- (5) The [Project Pane](#)
- (6) The [Watch Pane](#)
- (7) The [Output Pane](#)
- (8) The [Call Stack Pane](#)
- (9) The [Status Bar](#)

Operation Modes

Essentially, the TIDE GUI has two primary modes: The Edit Mode, in which you write the code for your application, and the Debug Mode, which is used while your code is running on the target and you are debugging it.

In each of these modes you may show, hide or resize various interface elements. The changes you make in one mode do not affect the other state. I.e, you could display the [Debug Toolbar](#) while in Debug Mode and hide it while in Edit Mode. Every time you will go into Debug Mode, the toolbar would appear. When you switch back to the Edit Mode, the toolbar will disappear. Thus, you may customize

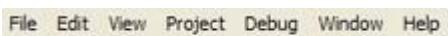
your workspace so it would serve you best both while editing code and while debugging.

Another key difference is that while in Debug Mode you cannot enter any new code. Whenever you try to type code while in this mode, you will be prompted to stop program execution on the target and switch back to Edit Mode.

You can easily tell the two modes apart by the background color of the code editor pane: When the background is white, you are in Edit Mode. When it's grey, you are in Debug Mode.

Menu Bar

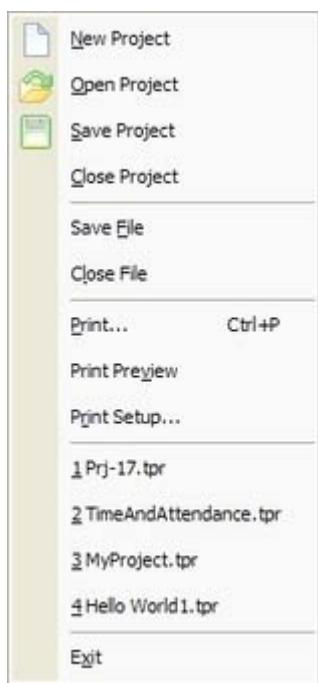
A standard bar, modeled after the classic Windows menu bar.



See below:

- [File Menu](#)^[115]
- [Edit Menu](#)^[116]
- [View Menu](#)^[117]
- [Project Menu](#)^[117]
- [Debug Menu](#)^[118]
- [Image Menu](#)^[119] (visible only when graphical resource is selected for editing)
- [Window Menu](#)^[119]
- [Help Menu](#)^[120]

File Menu^[3.1]



New Project: Displays the [New Project](#)^[127] dialog.

Open Project: Opens an existing project

Save Project: Saves all modified files on this project. Happens automatically on

build.

Close Project: Closes current project.

Save File: Saves the current file.

Close File: Closes the current file.

Print: Prints the current file.

Print Preview: Check output before printing.

Print Setup: Configure printing.

1... 4: Recent files.

Exit: Quits TIDE.

Edit Menu .3.2



Undo: Cancels last action.

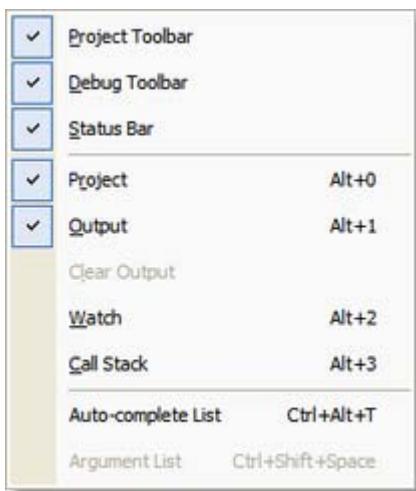
Redo: Cancels last undo.

Cut, Copy, Paste: Standard edit actions.

Select All: Select all text.

Find, Find Next, Find Prev, Find in Files, Replace: Standard find & replace actions.

View Menu 3.3



Project Toolbar: Toggles (shows/hides) the [Project](#) [120] toolbar.

Debug Toolbar: Toggles the [Debug](#) [121] toolbar.

Status Bar: Toggles the [Status Bar](#) [126].

Project: Toggles the [Project](#) [129] pane.

Output: Toggles the [Output](#) [129] pane.

Clear Output: Clears the contents of the Output pane.

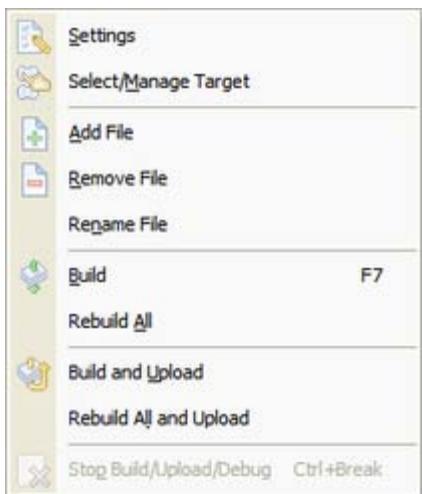
Watch: Toggles the [Watch](#) [130] pane.

Call Stack: Toggles the [Call Stack](#) [128] pane.

Auto-complete List: Show an appropriate [auto-complete list](#) [23] for the current context.

Argument List: Show a list of arguments for the current procedure call.

Project Menu 4



Settings: Displays the [Project Settings](#) [127] dialog.

Select/Manage Target: Shows the platform-specific dialog used to select a target for your project and upload firmware.

Add File: Displays the [Add file to project](#) dialog.

Remove File: Removes the file currently selected in the Project Tree from the project. Does not delete file from disk.

Rename File: Renames the file currently selected in the Project Tree.

Build: Builds project without uploading. Builds only files modified since last build.

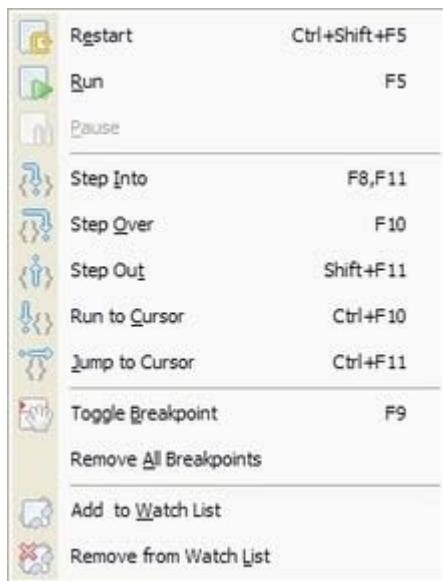
Rebuild All: Rebuilds project including all files -- even those not modified since last build.

Build and Upload: Builds if necessary, and uploads project to target without running it.

Rebuild All and Upload: Rebuilds project including all files, uploads and does not run.

Stop Build/Upload/Debug: Stops building, uploading or debugging. Exits debug mode. If the target was running, it will continue running.

Debug Menu⁵



Restart: Reboots the target device. Rebooting the device will not resume execution. Once the device has finished the reboot process, execution will be paused, pending further debug instructions.

Run: Begins or resumes program execution on target. If switching from Edit mode, this optionally compiles and uploads the project (when needed).

Pause: Pauses program execution. Covered under [Target States](#) above.

Step Into: Covered under [Stepping](#) above.

Step Over: Covered under [Stepping](#) above.

Step Out: Covered under [Stepping](#) above.

Run to Cursor: Covered under [Stepping](#) above.

Jump to Cursor: Covered under [Stepping](#) above.

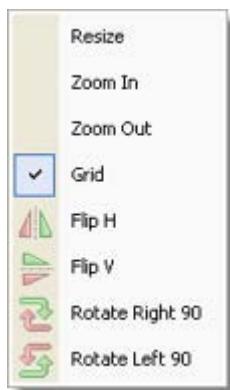
Toggle Breakpoint: Covered under [Breakpoints](#) above.

Remove All Breakpoints: Covered under [Breakpoints](#) above.

Add to Watch List: Covered under [The Watch](#) above.

Remove from Watch List: Covered under [The Watch](#) above.

Image Menu .6



This menu is visible only when a graphical resource is selected for editing. TIDE "knows" how to work with *.bmp*, *.jpg*, *.gif*, and *.png* files.

Resize: Changes the size of the image's "canvas". This is not re-sampling of the image: existing image elements won't shrink or get larger, just the total image size in pixels will change. If the new image size is smaller than portions of the image at the right and on the bottom will be lost.

Zoom In: Selects higher magnification. x1, x2, x4, x8, x16 magnification levels are available.

Zoom Out: Selects lower magnification. x1, x2, x4, x8, x16 magnification levels are available.

Grid: Toggles image grid visible/invisible.

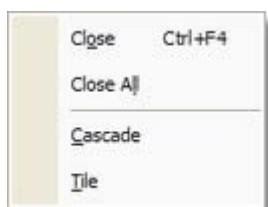
Flip H: Mirrors entire image or selected rectangular area (if selection is made) horizontally.

Flip V: Mirrors entire image or selected rectangular area (if selection is made) vertically.

Rotate Right 90: Rotates entire image or selected rectangular area (if selection is made) 90 degrees clockwise.

Rotate Left 90: Rotates entire image or selected rectangular area (if selection is made) 90 degrees counter-clockwise.

Window Menu



Close: Closes current window. Can also be done with Ctrl+W.

Close All: Closes all open documents without closing projects.

Cascade, Tile: Standard window actions.

Help Menu 3.8



Contents: Opens the help file at the contents.

Index: Opens the help file at the index.

Search: Opens the help file in search mode.

Context-Sensitive Help: Opens the help file at the topic for the currently selected keyword in the code editor.

Online Support: Open <http://www.tibbo.com/taiko.php> using the default browser.

About TIDE: Displays version information, etc.

Toolbars

The TIDE has the following toolbars:

- [Project Toolbar](#)^[120]
- [Debug Toolbar](#)^[121]
- [Image Editor Toolbar](#)^[122] (visible only when graphical resource is selected for editing)
- [Tool Properties Toolbar](#)^[122] (visible only when graphical resource is selected for editing, contents depend on the selected tool)

Project Toolbar



New Project: Displays the [New Project](#)^[127] dialog.



Open Project: Opens an existing project



Save Project: Saves all modified files on this project. Happens automatically on build.



Add File: Displays the [Add file to project](#)^[128] dialog.



Remove File: Removes current file from project. Does not delete file from disk.



Settings: Displays the [Project Settings](#)^[127] dialog.



Copy: Self-explanatory.



Cut: Self-explanatory.



Paste: Self-explanatory.



Undo: Cancels last action.



Redo: Cancels last undo.



Find: Self-explanatory.



Find Next: Self-explanatory.



Find Prev: Self-explanatory.



Replace: Self-explanatory.

Debug Toolbar



Select/Manage Target: Shows the platform-specific dialog used to select a target for your project and upload firmware.



Build: Builds project without uploading. Builds only files modified since last build.



Build and Upload: Uploads project to target without running it. Builds prior to upload, if required.



Restart: Reboots the target device. Rebooting the device will not resume execution. Once the device has finished the reboot process, execution will be paused, pending further debug instructions.



Run: Begins or resumes program execution on target. If switching from Edit mode, this optionally compiles and uploads the project (when needed).



Pause: Pauses program execution. Covered under [Target States](#) 28 above.



Stop Build/Upload/Debug: Stops building, uploading or debugging. Exits debug mode. If the target was running, it will continue running.



Step Into: Covered under [Stepping](#) 32 above.



Step Over: Covered under [Stepping](#) 32 above.



Step Out: Covered under [Stepping](#) 32 above.



Run to Cursor: Covered under [Stepping](#) 32 above.



Jump to Cursor: Covered under [Stepping](#) 32 above.



Toggle Breakpoint: Covered under [Breakpoints](#) 30 above.



Add to Watch List: Covered under [The Watch](#) 33 above.



Remove from Watch List: Covered under [The Watch](#) ^[33] above.

Image Editor Toolbar



This toolbar is visible only when graphical resource is selected for editing. When you select a certain tool an additional [Tool Properties Toolbar](#) ^[122] specifically for this tool is displayed. This toolbar provides all options for the selected tool.



Selection Tool: Selects rectangular image area to move, copy, or transform.



Hand Tool: Allows you to scroll the image using "hand grip" (position the tool over the image, click and hold left mouse button, then drag the image).



Paint Tool: Use this tool to do "freehand" painting over the image.



Eraser Tool: Erases portions of the image (paints with background color).



Text Tool: Adds text to the image.



Line Tool: Draws straight lines.



Rectangle Tool: Draws rectangles.



Ellipse Tool: Draws ellipses.



Eyedropper Tool: "Picks" the color off the image (left-click over the image to pick the fore-color; right click to pick the background color).



Zoom Tool: Changes the zoom level (magnification). x1, x2, x4, x8, x16 zoom levels are available.

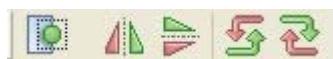
Tool Properties Toolbar

When you select a certain [image edit tool](#) ^[122] an additional properties toolbar specifically for this tool is displayed. This toolbar provides all options for the selected tool. The following toolbars are available:

- [Selection Tool Properties Toolbar](#) ^[123]
- [Paint Tool Properties Toolbar](#) ^[123]
- [Eraser Tool Properties Toolbar](#) ^[123]
- [Text Tool Properties Toolbar](#) ^[123]
- [Line Tool Properties Toolbar](#) ^[124]
- [Rectangle Tool Properties Toolbar](#) ^[124]

- [Ellipse Tool Properties Toolbar](#)^[125]
- [Zoom Tool Properties Toolbar](#)^[126]

Selection Tool Properties



Transparency Mode: For drag/drop and copy/paste operations selects whether pixels of background color will be copied to the destination as well. When the transparency mode is OFF the rectangular image fragment being copied or moved will overlay original image underneath completely. When the transparency mode is ON the original image will still be visible through the copied data.



Flip Horizontally: Mirrors entire image or selected rectangular area (if selection is made) horizontally.



Flip Vertically: Mirrors entire image or selected rectangular area (if selection is made) vertically.

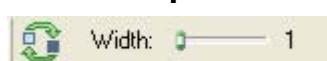


Rotate Left 90: Rotates entire image or selected rectangular area (if selection is made) 90 degrees counter-clockwise.



Rotate Right 90: Rotates entire image or selected rectangular area (if selection is made) 90 degrees clockwise.

Paint Tool Properties

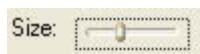


Inversion Mode: Toggles paint tool's inversion mode on/off. When inversion is OFF, left-clicking on an image pixel changes this pixel's color to fore-ground color; right-clicking changes the color to background color. With inversion ON, left-clicking causes the pixel to alternate between fore-ground and background colors.



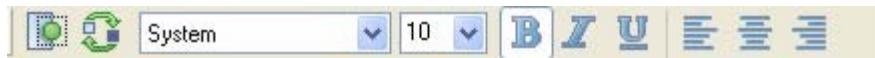
Size: "Pencil tip" width in pixels.

Eraser Tool Properties



Size: "Eraser width" in pixels.

Text Tool Properties





Transparency Mode: selects whether original image will still be visible underneath the text. When the transparency mode is OFF, original image will not be visible. When the transparency mode is ON, original image will be visible.



Inversion Mode: Toggles text tool's inversion mode on/off. When inversion is OFF, the text will always be printed in the fore-ground color. When inversion is ON, the text will appear in fore-ground color over the areas originally painted in background color and in background color over the areas originally painted with non-background color.



Font Selector: Selects the font to print the text with.



Font Size: Selects the size of the font.



Bold: Toggles bold attribute for the font on/off.



Italic: Toggles italic attribute for the font on/off.



Underline: Toggles underline attribute for the font on/off.



Left: Selects left alignment for the text (result is visible with multi-line text only).

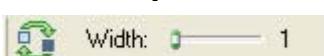


Center: Selects center alignment for the text (result is visible with multi-line text only).



Right: Selects right alignment for the text (result is visible with multi-line text only).

Line Tool Properties



Inversion Mode: Toggles line tool's inversion mode on/off. When inversion is OFF, the line will always be drawn in the fore-ground color. When inversion is ON, the line will appear in fore-ground color over the areas originally painted in background color and in background color over the areas originally painted with non-background color.



Size: Line width in pixels.

Rectangle Tool Properties





Inversion Mode: Toggles rectangle tool's inversion mode on/off. When inversion is OFF, the rectangle's border will always be drawn in the fore-ground color and filling (for filled rectangles) will always be done with background color. When inversion is ON, the rectangle's border will appear in fore-ground color over the areas originally painted in background color and in background color over the areas originally painted with non-background color. The color for the "filling" (when enabled) will be exactly opposite.



Solid Rectangle: Creates a rectangle filled with fore-ground color (+ inversion mode effect is applied with enabled).



Filled Rectangle: Creates a rectangle with the border of fore-ground color and filled with the background color (+ inversion mode effect is applied with enabled).



Unfilled Rectangle: Creates a rectangle with the border of fore-ground color and no filling (+ inversion mode effect is applied with enabled).



Size: Border width (irrelevant for solid rectangles).



Corner rounding: Defines the radius of rectangle corners.

Ellipse Tool Properties



Inversion Mode: Toggles ellipse tool's inversion mode on/off. When inversion is OFF, the ellipse's border will always be drawn in the fore-ground color and filling (for filled ellipses) will always be done with background color. When inversion is ON, the ellipse's border will appear in fore-ground color over the areas originally painted in background color and in background color over the areas originally painted with non-background color. The color for the "filling" (when enabled) will be exactly opposite.



Solid Ellipse: Creates an ellipse filled with fore-ground color (+ inversion mode effect is applied with enabled).



Filled Ellipse: Creates an ellipse with the border of fore-ground color and filled with the background color (+ inversion mode effect is applied with enabled).



Unfilled Ellipse: Creates an ellipse with the border of fore-ground color and no filling (+ inversion mode effect is applied with enabled).



Size: Border width (irrelevant for solid ellipses).

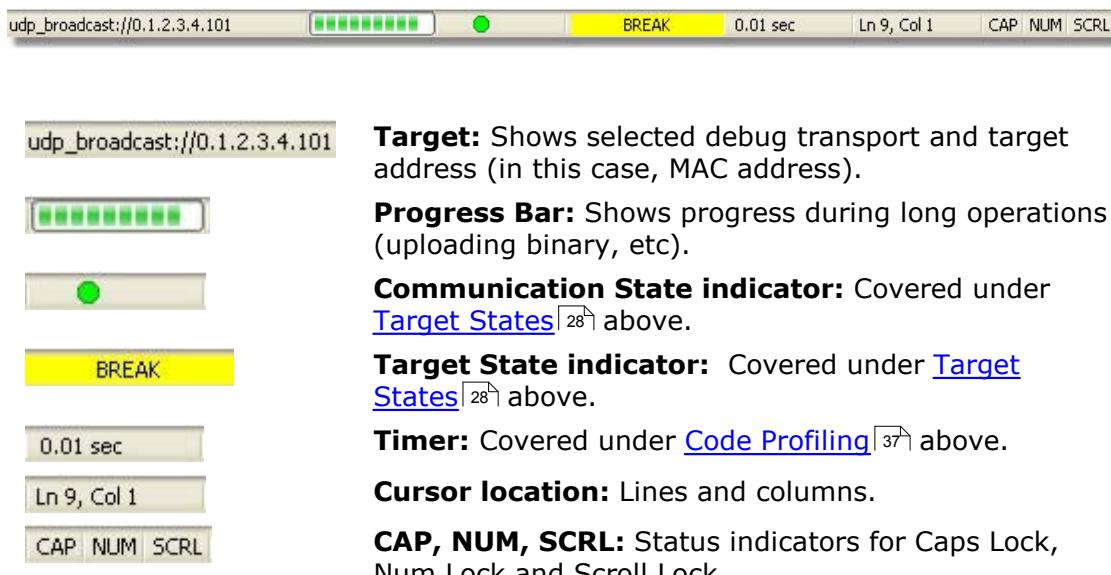
Zoom Tool Properties



Zoom Level: Selects one of available zoom (magnification) levels.

Status Bar

Below is a screenshot of the status bar:



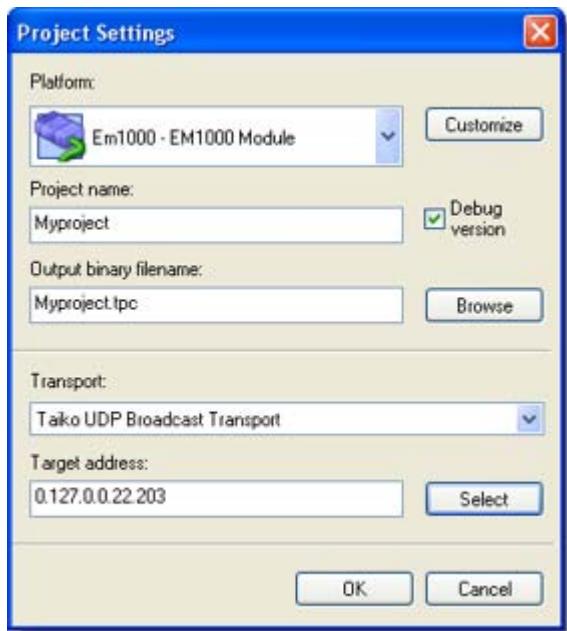
Dialogs

Not all dialogs are reviewed -- only the ones which are not self-explanatory.

In this section:

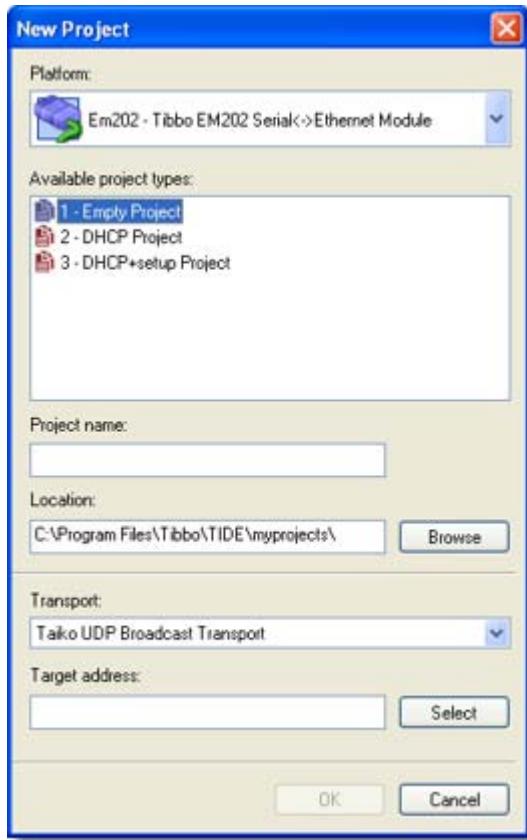
- [Project Settings](#)^[127]
- [New Project](#)^[127]
- [Add file to Project](#)^[128]
- [Graphic File Properties Dialog](#)^[128]

Project Settings



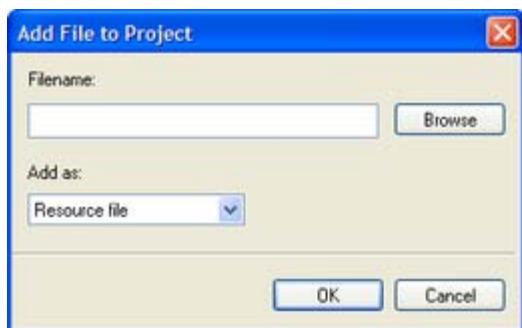
This dialog has been covered under [Project Settings](#)³⁸ above.

New Project.2



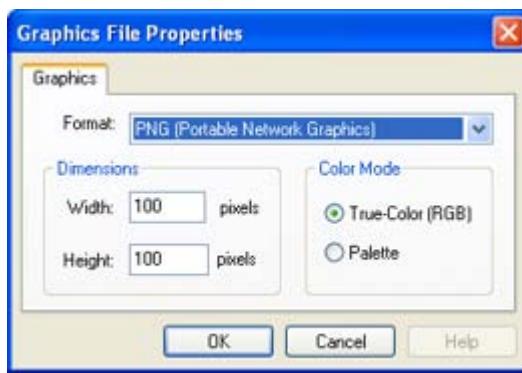
This dialog has been covered under [Starting a New Project](#)¹⁰ above.

Add File to Project



This dialog has been covered under [Adding, Removing and Saving Files](#)¹⁸ above.

Graphic File Properties Dialog



This dialog has been covered under [Adding, Removing and Saving Files](#)¹⁸ above.

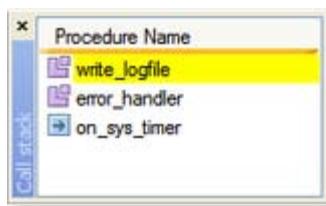
Panes

Some panes may be toggled using shortcut keys or the [View Menu](#)¹¹⁷. [Colors pane](#)¹³⁰ is displayed automatically when an image resource file is opened for editing.

In this section:

- [Call Stack](#)¹²⁸
- [Output](#)¹²⁹
- [Project](#)¹²⁹
- [Watch](#)¹³⁰
- [Colors](#)¹³⁰

Call Stack.7.1



The Call Stack pane is covered under [The Call Stack](#)³¹ above.

Output

6.2.7.2



```
Building...
Compiling main.bas...
ggg.html skipped...
Linking...
Application file size: 384 bytes
Required variable memory (RAM) size: 11 bytes
Build complete.
Uploading...
Application file loaded.
Debug session started...
```

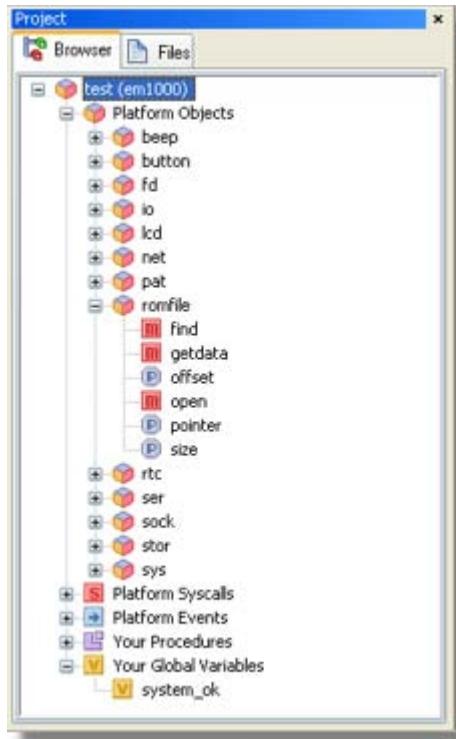
Displays status messages while compiling, linking, uploading and debugging.
Double clicking on an error message would move the cursor to the line of code which caused the error.

Project

6.2.7.3

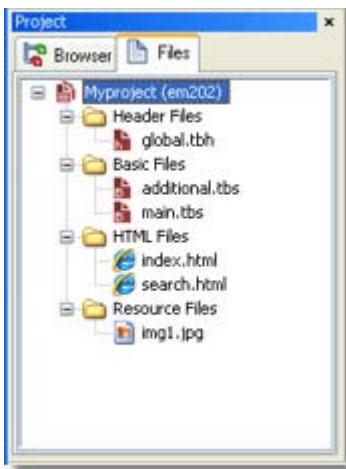
The Project pane contains two tabs: [Browser](#)¹²⁹ and [Files](#)¹³⁰.

Browser



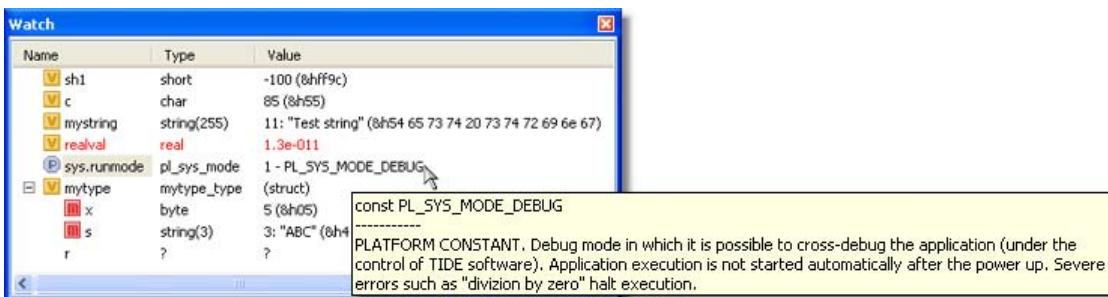
The Browser tab has been covered under [Using the Project Browser](#)²² and under [The Watch](#)³³ above.

Files



The Files tab has been covered under [Adding, Removing and Saving Files](#)¹⁸ above.

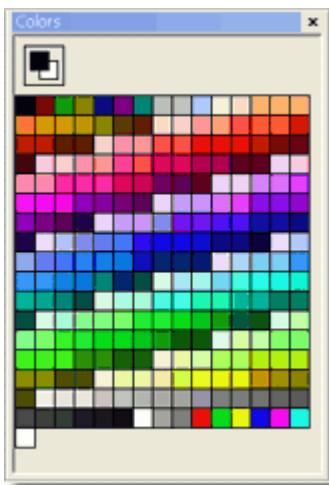
Watch 6.2.7.4



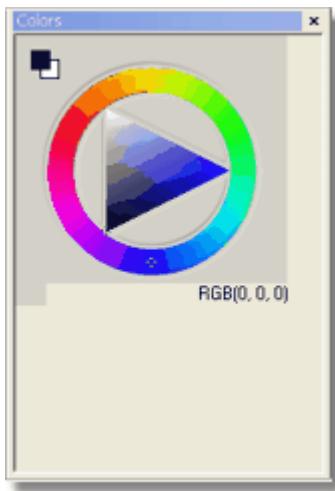
The Watch pane has been covered under [The Watch](#)³³ above.

Colors 6.2.7.5

Colors pane is displayed whenever a graphical resource is opened for editing in TIDE. Depending on the color mode selection you've made when adding an image file to the project, the pane will either show available palette colors...



... or RGB color selector:



Language Element Icons

Throughout TIDE, many icons are used for various Tibbo Basic constructs. Below is a complete listing:

- Constants** (see [Constants](#))
- Enumeration Types** (see [User-Defined Types](#))
- System Calls** (see [Function Reference](#))
- Objects** (see [Object Reference](#))
- Properties** (see [Object Reference](#))
- Methods** (see [Object Reference](#))
- Event Handlers** (Implemented in current project -- see [Object Reference](#)). Grayed if no event handler exists for an event.
- Procedures** (see [Introduction to Procedures](#)). Grayed if a procedure is not implemented (i.e. doesn't have a body).
- Variables** (see [Introduction to Variables, Constants and Scopes](#)). Grayed if the variable is not defined.

Glossary of Terms

Below are several key definitions for terms used throughout the text.

Compilation Unit

A single file containing source code, to be processed by the [compiler](#). Projects may contain many compilation units. Under Tibbo Basic, BASIC source files (.tbs) and HTML files (.html) are compilation units.

Compiler

For Tibbo Basic, a software program which takes compilation units and converts each of them, individually, to executable [P-Code](#). The Tibbo Basic compiler is a single-pass compiler, which means it goes over each compilation unit from beginning to end, and does it just one time.

Construct

A meaningful combination of language elements, including keywords, identifiers, constants, etc. An example of a construct would be `A = B + 5`.

Cross-Debugging

This is the practice of using one device to observe and control the state of a program running on another device, in order to find bugs in it. Under Tibbo Basic, your computer displays various status messages and information about variables etc, but the actual code is executed on the [target](#)^[133]. The target state is periodically polled and displayed on your computer. Hence, cross-debugging.

Identifier

Any 'name' for a variable, a function, a subroutine, a constant, or any other 'thing' you may call or refer to within a program. In the statement `x = 5`, `x` is an identifier. An existing [keyword](#)^[132] cannot be used as an identifier, since it already has a fixed meaning as part of Tibbo Basic syntax.

Keyword

A single word which carries a specific meaning within Tibbo Basic. Keywords are listed under [Keywords](#)^[96] above.

Label

An [identifier](#)^[132] marking the beginning of a block of code which will then be called using a [Goto Statement](#)^[88]. Labels are declared in code by writing their name, followed by a colon, on a single line.

Linker

A software program processing the output of the [compiler](#)^[131], to look for any cross-references between the units. If compilation unit A calls a procedure which is in compilation unit B, the linker associates between the two, and provides compilation unit A with the proper memory addresses so that it could actually reach the procedure it needs in unit B.

P-Code

Pseudo-Code. This is code which is not executed directly by a processor, but by a 'virtual processor' (called a Virtual Machine) which is a part of TiOS that emulates a processor, interprets the P-Code and executes it.

The Tibbo Basic compiler produces P-Code.

Syscall

A system *call*. This is an internal platform function -- not to be used explicitly. It is expressed as a numeric value. Syscalls are automatically invoked when you perform certain operations in code, such as variable type conversion.

You do not have to invoke syscalls directly within your code -- it is not recommended.

Target

The hardware device with which you are working. This is the device connected to the computer while debugging. The code you are writing actually runs on this device, and the debug messages originate from the device -- not from anywhere within your computer.

Virtual Machine

This is a part of TiOS. In essence, it is a processor implemented in software. It executes the [P-Code](#)^[132] of your application (produced by the compiler).

Using a Virtual Machine, we can achieve full control over its code execution. You can think of your application as if it runs in a designated 'sandbox' -- you can do anything, but the Operating System will stay unharmed. So no code executed in the Virtual Machine can crash TiOS itself.

This approach also greatly enhances your control over your program execution during debugging.

Platforms

This section contains specifications for all platforms included into the current documentation.

Each platform supports a number of functions (syscalls) and objects. Actual functions and object descriptions are not included into each platform's spec. Instead, they are documented in the [Function Reference](#)^[189] and [Object Reference](#)^[212] sections, while [Platform Specifications](#)^[133] section only contains the lists of functions and objects supported. This is because most functions and objects are shared by different platforms.

Platform Specifications

The following platforms are included into this documentation:

Platform	Devices
EM202 ^[134]	EM200, EM203, DS203
EM1000 ^[139]	EM1000, DS1000
EM1000W ^[139]	EM1000 + GA1000
EM1202 ^[149]	EM1202, EM1202EV*, DS1202*
EM1202W ^[149]	EM1202 + GA1000
DS1202 ^[166]	EM1202EV*, DS1202*
EM1206 ^[158]	EM1206
EM1206W ^[158]	EM1206 + GA1000
DS1206 ^[174]	DS1206, DS1206N

* These devices can be used with the EM1202 or DS1202 platform. Notice, however, that the EM1202EV and DS1202 interconnect certain pins of the EM1202. See [Programmable Hardware Manual](#) for details.

EM202 Platform

- [Memory Space](#)^[134]
- [Supported Variable Types](#)^[134]
- [LED signals](#)^[184] (common for all devices)
- [Debug communications](#)^[185] (common for all devices)
- [Project Settings Dialog](#)^[186] (common for all devices)
- [Supported Functions](#)^[134]
- [Supported Objects](#)^[135]
- [Platform-dependent Constants](#)^[135]
- [Platform-dependent Programming Information](#)^[136]

Memory Space

This platform has the following amounts of program (FLASH) and variable (RAM) memory available:

Program memory (FLASH): 65,408 bytes

Variable memory (RAM): 20,480 bytes

EEPROM memory (STOR): 2048 bytes, of which 8 bytes are occupied by MAC address of the device*

*See [Platform-dependent programming information](#)^[136] for details.

Supported Variable Types

This platform supports the following variable types:

- Byte.
- Word.
- Char.
- Short (integer).
- Boolean.
- User-defined structures.
- User-defined enumeration types.

For general type description see [Variables and Their Types](#)^[43].

Supported Functions (Syscalls)

The following syscalls (platform functions) are supported by the platform:

- Conversion to and from strings:
 - [Asc](#)^[189] string character --> ASCII code;
 - [Chr](#)^[191] ASCII code --> string character;

- [Val](#)^[210] numerical string--> 16-bit value (word or short);
- [Bin](#)^[190] unsigned 16-bit numeric value (word) --> binary numerical string;
- [Str](#)^[207] unsigned 16-bit numeric value (word) --> decimal numerical string;
- [Stri](#)^[208] signed 16-bit numeric value (short) --> decimal numerical string;
- [Hex](#)^[195] unsigned 16-bit numeric value (word) --> hexadecimal numerical string;
- [Len](#)^[198] gets the string length;
- [Left](#)^[198] gets a left portion of a string;
- [Mid](#)^[202] gets a middle portion of a string;
- [Right](#)^[205] gets a right portion of a string;
- [Instr](#)^[197] finds a substring in a string;
- [Strgen](#)^[207] generates a string using repeating substring;
- [Strsum](#)^[209] calculates 16-bit (word) sum of string characters' ASCII codes.

Supported Objects

The following objects are found on this platform:

- Sockets ([sock](#)^[274].) object -- supports up to 16 simultaneous UDP or TCP connections, or HTTP sessions (excludes [sock.redir](#)^[346], [sock.allowedinterfaces](#)^[327], [sock.targetinterface](#)^[359], [sock.currentinterface](#)^[331]);
- Ethernet ([net](#)^[267].) object -- controls Ethernet interface;
- Serial ([ser](#)^[224].) object -- supports 4 serial channels; each channel can work in the UART, Wiegand, and clock/data modes (excludes [ser.redir](#)^[260]);
- Input/output ([io](#)^[365].) object -- handles I/O lines, ports, and interrupts (only includes [io.num](#)^[372] and [io.state](#)^[375]);
- EEPROM ([stor](#)^[380].) object -- facilitates access to the EEPROM memory (excludes [offset](#)^[378]);
- ROM data ([romfile](#)^[375].) object -- provides access to the fixed ("ROM") data of your Tibbo BASIC application;
- LED pattern ([pat](#)^[384].) object -- "plays" patterns on Green and Red Status LEDs;
- System button ([button](#)^[272].) object -- handles special system (MD) button;
- System ([sys](#)^[212].) object -- controls general device functionality (excludes [sys.onsystimerperiod](#)^[220], [sys.serialnum](#)^[221], [sys.setserialnum](#)^[222]).

Platform-dependent Constants

The following constant lists are platform-specific:

- [Enum_pl_redir](#)^[136]- a list of constants that define buffer redirection (shorting) for this platform.
- [Enum_pl_io_num](#)^[136]- a list of constants that define available I/O lines.

Enum pl_redir

Starting from V2.0, the EM202 platform no longer supports redirection ([sock.redir](#)³⁴⁵ and [ser.redir](#)²⁶⁰ methods have been removed because of space constraints). Therefore, pl_redir constant has been deleted.

Enum pl_io_num

Enum pl_io_num contains the list of constants that refer to I/O lines available on this platform. Use these constants when selecting the line with the [io](#)³⁶⁵ object (see the [io.num](#)³⁷² property).

Enum pl_io_num for this platform includes the following constants:

- | | |
|---------------------|--|
| 0- PL_IO_NUM: | Selects general-purpose I/O line 0. Only available on the EM200. |
| 1- PL_IO_NUM_1: | Selects general-purpose I/O line 1. |
| 2- PL_IO_NUM_2_DSR: | Selects general-purpose I/O line 2. This is also a DSR input of serial port 0. On EM202 and EM200 works as input/output. On EM202-EV and DS202 works as input only and the actual state of DSR pin is opposite to what is read through the io.state property. |
| 3- PL_IO_NUM_3_DTR: | Selects general-purpose I/O line 3. This is also a DTR output of serial port 0. On EM202 and EM200 works as input/output. On EM202-EV and DS202 works as output only and the actual state of DTR pin is opposite to what is set through the io.state property. |
| 4- PL_IO_NUM_4_CTS: | Selects general-purpose I/O line 4. This is also a CTS input of serial port 0. On EM202 and EM200 works as input/output. Cannot be set through io.state when ser.flowcontrol= 1- ENABLED. On EM202-EV and DS202 works as input only and the actual state of CTS pin is opposite to what is read through the io.state property. |
| 5- PL_IO_NUM_5_RTS: | Selects general-purpose I/O line 5. Only available on the EM200. This is also a RTS output of serial port 0. On EM202 and EM200 works as input/output. Cannot be set through io.state when ser.flowcontrol= 1- ENABLED. On EM202-EV and DS202 works as output only and the actual state of RTS pin is opposite to what is set through the io.state property. |
| 6- PL_IO_NUM_6: | Selects general-purpose I/O line 6. Only available on the EM200. |
| 7- PL_IO_NUM_7: | Selects general-purpose I/O line 7. Only available on the EM200. |
| 8- PL_IO_NUM_8: | Selects general-purpose I/O line 8. Only available on the EM200. |

Platform-dependent Programming Information

This section contains miscellaneous information that sets this platform apart from other platforms. Various objects described in the [Object Reference](#)²¹² direct you to this topic in all cases when object capabilities or behavior depends on the platform

and, hence, cannot be described in the object reference itself. Each platform section in this manual has its own "Platform-dependent Programming Information" topic. If you are reading documentation top-to-bottom (we have never actually met anyone who does) you can skip this section now.

Supported variable types

This platform supports the following variable types^[43]: byte, word, char, short (integer), string, boolean, user-defined structures and enumeration types. Dword, long, and real (float) types are not supported.

There is no need to explicitly configure I/O lines as inputs or outputs

On this platform, each I/O line (controlled by the [io](#)^[365] object) can be an input and an output at the same time (such lines are sometimes called "quasi-bidirectional"). This is why on this platform the [io](#) object does not have [io.enabled](#)^[370] property.

To sense the state of the external signal applied to the I/O line set the line to HIGH first:

```
...
io.num= PL_IO_NUM  'just to select some line as an example
io.state= HIGH      'now we can read the line
x=io.state          'read line state into x
...
```

I/O lines are not "afraid" of "signal competition". That is, you won't damage the line if it is outputting HIGH while external signal is driving it LOW, or vice versa. If the line is driven LOW internally or externally, resulting state of the line is LOW.



For the hardware-savvy reader: output drivers of I/O lines on this platform are implemented as open collector output with weak pull-up resistor (~20K).

Remapping of I/O lines of the serial port is not possible

All lines of the [serial port](#)^[224]- **TX/W1out/dout output, RX/W1in/din input, RTS/W0out/cout output, and CTS/W0&1in/cin input** have fixed positions and cannot be reassigned to different I/O pins of the device. Therefore, on this platform the [ser](#)^[224] object does not have [ser.ctsmap](#)^[249] and [ser.ctsmap](#)^[249] properties.

Explicit configuration of the I/O lines of the serial port as inputs or outputs is not required

Since all I/O lines can serve as inputs and outputs at the same time it is not necessary to explicitly configure [serial port](#)^[224] lines as inputs or outputs.

Serial port does not have a FIFO buffer

When the [serial port](#)^[224] is in the [UART/full-duplex/flow control](#)^[226] mode ([ser.mode](#)^[255]= [PL_SER_MODE_UART](#), [ser.interface](#)^[255]= [PL_SER_SI_FULLDUPLEX](#), and [ser.flowcontrol](#)^[253]= 1- ENABLED) the device is monitoring its CTS input to see if attached serial device is ready to receive more data. If the CTS state changes to "cannot transmit" the device will stop sending out data immediately. Outgoing serial character that has already started transmitting will be sent out, but no more

characters will be sent until the CTS line state changes to "can transmit".

There is no PLL

On this platform, there is no PLL and device operating speed is always the same. Therefore, on this platform the [sys](#)^[212] object does not have [sys.currentpll](#)^[218] read-only property and [sys.newpll](#)^[219] method.

Data in the special configuration section of the EEPROM

Bottom 8 bytes of the EEPROM (accessible through the [stor](#)^[380] object) are reserved for storing MAC address of the device. On power-up, the MAC address is read out from the EEPROM and programmed into the Ethernet controller. You can always check current MAC through the [net.mac](#)^[269] read-only property of the [net](#)^[267] object but there is no direct way to change it. Instead, you can change the MAC address data in the EEPROM. Then, next time the device boots up it will start using this new address.

By default, the area storing MAC address is not accessible to your application- the [stor.base](#)^[381] property takes care of that. Unless you change it, this property specifies that your application's storage area starts at address 9 (counting from 1). To change MAC, set the stor.base to 1.

MAC address data in the EEPROM has a certain formatting -- you have to follow it if you want the MAC to be recognized by the firmware (TiOS). Here is the format:

Byte1	Byte2	Byte3	Byte4	Byte5	Byte 6	Byte7	Byte8
6	MAC0	MAC1	MAC2	MAC3	MAC 4	MAC5	Checksum

Byte at address 1 must be set to 6- this means, that 6 byte of data follow (MAC address consists of 6 bytes). Addresses from 2 to 7 carry actual MAC data. Address 8 stores the checksum, which is calculated like this:

$255 - (\text{modulo}_8 \text{ sum of addr_1_through_7})$

Here is a sample code that stores new MAC address into the EEPROM and then reboots the device to make the device load this new MAC:

```

dim s as string
dim x as byte
...
s= "0.2.123.124.220.240" 'supposing, we want to set this MAC
...
...
s=chr(6)+ddval(s)      'added first byte (always 6) and converted
readable MAC into bytes
x=255-strsum(s)        'calculated checksum and assigned the
result to a BYTE variable (!!!)
s=s+chr(x)              'now our string is ready

stor.base(1)             'will access EEPROM from the bottom
x=stor.set(s,1)           'save data
if x<>len(s) then       'it is a good programming practice to check the
result
    'failed
else
    sys.reboot            'new MAC set, reboot!
end if

...

```

There are limitations on what MAC you can set. When loading the MAC into the Ethernet controller the device always resets the first byte of this address to 0. For example, if you set the MAC to 1.2.3.4.5.6 then the actual MAC used by the device will be 0.2.3.4.5.6.



If you write incorrect MAC data (wrong first byte or error in checksum calculation) the device will ignore it and boot up with default MAC, which is 0.1.2.3.4.100.

Available network interfaces

This platform only has one network interface -- the Ethernet port.

Miscellaneous

- [Sys](#)^[212] object does not support the [sys.onSystemPeriod](#)^[220] property and the [on sys timer](#)^[220] event generation period is fixed at 0.5 seconds.
- Beginning with **V2.0**, [sock.redirect](#)^[346] and [ser.redirect](#)^[260] methods are no longer supported by this platform.
- Also beginning with **V2.0**, the following functions are no longer supported: [date](#)^[191], [dayCount](#)^[192], [hours](#)^[195], [minuteCount](#)^[203], [minutes](#)^[203], [month](#)^[204], [weekday](#)^[211], [year](#)^[211].

EM1000 and EM1000W Platforms

The difference between the EM1000 and EM1000W platforms is that the EM1000W additionally includes the [wln](#)^[497] (Wi-Fi) object, which works with an external GA1000 add-on module to function. All other features of these two platforms are exactly the same. Both platforms will be collectively referred to as "EM1000(W)".

- [Memory Space](#)^[140]
- [Supported Variable Types](#)^[183] (common for T1000-based devices)

- [LED signals](#)^[184] (common for all devices)
- [Debug communications](#)^[185] (common for all devices)
- [Project Settings Dialog](#)^[186] (common for all devices)
- [Supported Functions](#)^[183] (common for T1000-based devices)
- [Supported Objects](#)^[140]
- [Platform-dependent Constants](#)^[140]
- [Platform-dependent Programming Information](#)^[145]

Memory Space

The EM1000(W) has the following amounts of program (FLASH) and variable (RAM) memory available:

Program memory: 458,752 Bytes for the EM1000(W)-512K;
983040 Bytes for the EM1000(W)-1024K

Variable memory: 20,480 bytes

EEPROM memory: 2048 bytes, of which 8 bytes are occupied by MAC address of the device*

*See [Platform-dependent programming information](#)^[145] for details.

Supported Objects

The following objects are found on the EM1000(W):

- [Sock](#)^[274] — socket communications (up to 16 UDP, TCP, and HTTP sessions);
- [Net](#)^[267] — controls Ethernet port;
- [WIn](#)^[497] — handles Wi-Fi interface (only available on the **EM1000W platform**, requires GA1000 add-on module);
- [Ser](#)^[224] — in charge of serial ports (UART, Wiegand, and clock/data modes);
- [Io](#)^[365] — handles I/O lines, ports, and interrupts;
- [Lcd](#)^[392] — controls graphical display panels (several types supported);
- [Kp](#)^[484] — scans keypads of matrix and "binary" types;
- [Rtc](#)^[389] — keeps track of date and time;
- [Fd](#)^[433] — manages flash memory file system and direct sector access;
- [Stor](#)^[380] — provides access to the EEPROM;
- [Romfile](#)^[375] — facilitates access to resource files (fixed data);
- [Pat](#)^[384] — "plays" patterns on up to five LED pairs;
- [Beep](#)^[387] — generates buzzer patterns;
- [Button](#)^[272] — monitors MD line (setup button);
- [Sys](#)^[212] — in charge of general device functionality.

Platform-dependent Constants

The following constant lists are platform-specific:

- [Enum_pl_redir](#)^[141]- a list of constants that define buffer redirection (shorting) for this platform.
- [Enum_pl_io_num](#)^[142]- a list of constants that define available I/O lines.

- [Enum pl_int_num](#)^[144]- a list of constants that define available *interrupt* lines.
- [Enum pl_sock_interfaces](#)^[145]- a list of available network interfaces.

Enum pl_redir

Enum pl_redir contains the list of constants that define buffer redirection (shorting) for this platform. The following objects support buffers and buffer redirection on the EM1000(W):

- [Ser](#)^[224] object (see [ser.redir](#)^[260] method)
- [Sock](#)^[274] object (see [sock.redir](#)^[346] method)

Enum pl_redir for this platform includes the following constants:

0- PL_REDIR_NONE:	Cancels redirection for the serial port or socket.
1- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0. This constant can be used as a "base" for all other serial ports, i.e. in expressions like ser.redir= PL_REDIR_SER+f.
1- PL_REDIR_SER0:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0.
2- PL_REDIR_SER1:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 1.
3- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 2.
4- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 3.
6- PL_REDIR_SOCK0:	Redirects RX data of the serial port or socket to the TX buffer of socket 0. This constant can be used as a "base" for all other sockets i.e. in expressions like sock.redir= PL_REDIR_SOCK0+f.
7- PL_REDIR_SOCK1:	Redirects RX data of the serial port or socket to the TX buffer of socket 1.
8- PL_REDIR_SOCK2:	Redirects RX data of the serial port or socket to the TX buffer of socket 2.
9- PL_REDIR_SOCK3:	Redirects RX data of the serial port or socket to the TX buffer of socket 3.
10- PL_REDIR_SOCK4:	Redirects RX data of the serial port or socket to the TX buffer of socket 4.
11- PL_REDIR_SOCK5:	Redirects RX data of the serial port or socket to the TX buffer of socket 5.
12- PL_REDIR_SOCK6:	Redirects RX data of the serial port or socket to the TX buffer of socket 6.
13- PL_REDIR_SOCK7:	Redirects RX data of the serial port or socket to the TX buffer of socket 7.
14- PL_REDIR_SOCK8:	Redirects RX data of the serial port or socket to the TX buffer of socket 8.
15- PL_REDIR_SOCK9:	Redirects RX data of the serial port or socket to the TX buffer of socket 9.
16- PL_REDIR_SOCK10:	Redirects RX data of the serial port or socket to the TX

- buffer of socket 10.
- 17- PL_REDIR SOCK11: Redirects RX data of the serial port or socket to the TX buffer of socket 11.
- 18- PL_REDIR SOCK12: Redirects RX data of the serial port or socket to the TX buffer of socket 12.
- 19- PL_REDIR SOCK13: Redirects RX data of the serial port or socket to the TX buffer of socket 13.
- 20- PL_REDIR SOCK14: Redirects RX data of the serial port or socket to the TX buffer of socket 14.
- 21- PL_REDIR SOCK15: Redirects RX data of the serial port or socket to the TX buffer of socket 15.

Enum pl_io_num

Enum pl_io_num contains the list of constants that refer to I/O lines available on this platform. Use these constants when selecting the line with the [io](#)^[365] object (see the [io.num](#)^[372] property).

All I/O lines of the EM1000(W) require explicit configuration as inputs or outputs -- this is done through the [io.enabled](#)^[370] property of the [io](#)^[365] object. On power-up, all lines are configured as inputs. When the line is configured for input its output driver is tri-stated. When the line is configured for output, its output driver is enabled. It is possible to read the state of the line even when it is working as an output.

Certain lines automatically become inputs and outputs in certain modes of operation -- see below for details.

Enum pl_io_num includes the following constants:

- | | |
|---------------------------------------|--|
| 0- PL_IO_NUM_0: | General-purpose I/O line 0 (P0.0). |
| 1- PL_IO_NUM_1: | General-purpose I/O line 1 (P0.1). |
| 2- PL_IO_NUM_2: | General-purpose I/O line 2 (P0.2). |
| 3- PL_IO_NUM_3: | General-purpose I/O line 3 (P0.3). |
| 4- PL_IO_NUM_4: | General-purpose I/O line 4 (P0.4). |
| 5- PL_IO_NUM_5: | General-purpose I/O line 5 (P0.5). |
| 6- PL_IO_NUM_6: | General-purpose I/O line 6 (P0.6). |
| 7- PL_IO_NUM_7: | General-purpose I/O line 7 (P0.7). |
| 8- PL_IO_NUM_8_RX0 ⁽¹⁾ : | General-purpose I/O line 8 (P1.0). This line is also the RX/W1in/din ^[225] input of the serial port 0. |
| 9- PL_IO_NUM_9_TX0 ⁽²⁾ : | General-purpose I/O line 9 (P1.1). This line is also the TX/W1out/dout ^[225] output of the serial port 0. |
| 10- PL_IO_NUM_10_RX1 ⁽¹⁾ : | General-purpose I/O line 10 (P1.2). This line is also the RX/W0&1in/din input of the serial port 1. |
| 11- PL_IO_NUM_11_TX1 ⁽²⁾ : | General-purpose I/O line 11 (P1.3). This line is also the TX/W1out/dout output of the serial port 1. |
| 12- PL_IO_NUM_12_RX2 ⁽¹⁾ : | General-purpose I/O line 12 (P1.4). This line is also the RX/W0&1in/din input of the serial port 2. |
| 13- PL_IO_NUM_13_TX2 ⁽²⁾ : | General-purpose I/O line 13 (P1.5). This line is also the TX/W1out/dout output of the serial port 2. |
| 14- PL_IO_NUM_14_RX3 ⁽¹⁾ : | General-purpose I/O line 14 (P1.6). This line is also |

	the RX/W0&1in/din input of the serial port 3.
15- PL_IO_NUM_15_TX3 ⁽²⁾ :	General-purpose I/O line 15 (P1.7). This line is also the TX/W1out/dout output of the serial port 3.
16- PL_IO_NUM_16_INT0:	General-purpose I/O line 16 (P2.0).
17- PL_IO_NUM_17_INT1:	General-purpose I/O line 17 (P2.1).
18- PL_IO_NUM_18_INT2:	General-purpose I/O line 18 (P2.2).
19- PL_IO_NUM_19_INT3:	General-purpose I/O line 19 (P2.3).
20- PL_IO_NUM_20_INT4:	General-purpose I/O line 20 (P2.4).
21- PL_IO_NUM_21_INT5:	General-purpose I/O line 21 (P2.5).
22- PL_IO_NUM_22_INT6:	General-purpose I/O line 22 (P2.6).
23- PL_IO_NUM_23_INT7:	General-purpose I/O line 23 (P2.7).
24- PL_IO_NUM_24:	General-purpose I/O line 24 (P3.0).
25- PL_IO_NUM_25:	General-purpose I/O line 25 (P3.1).
26- PL_IO_NUM_26:	General-purpose I/O line 26 (P3.2).
27- PL_IO_NUM_27:	General-purpose I/O line 27 (P3.3).
28- PL_IO_NUM_28:	General-purpose I/O line 28 (P3.4).
29- PL_IO_NUM_29:	General-purpose I/O line 29 (P3.5).
30- PL_IO_NUM_30:	General-purpose I/O line 30 (P3.6).
31- PL_IO_NUM_31:	General-purpose I/O line 31 (P3.7).
32- PL_IO_NUM_32:	General-purpose I/O line 32 (P4.0).
33- PL_IO_NUM_33:	General-purpose I/O line 33 (P4.1).
34- PL_IO_NUM_34:	General-purpose I/O line 34 (P4.2).
35- PL_IO_NUM_35:	General-purpose I/O line 35 (P4.3).
36- PL_IO_NUM_36:	General-purpose I/O line 36 (P4.4).
37- PL_IO_NUM_37:	General-purpose I/O line 37 (P4.5).
38- PL_IO_NUM_38:	General-purpose I/O line 38 (P4.6).
39- PL_IO_NUM_39:	General-purpose I/O line 39 (P4.7).
40- PL_IO_NUM_40:	General-purpose I/O line 40 (does not belong to any 8-bit port).
41- PL_IO_NUM_41:	General-purpose I/O line 41 (does not belong to any 8-bit port).
42- PL_IO_NUM_42:	General-purpose I/O line 42 (does not belong to any 8-bit port).
43- PL_IO_NUM_43:	General-purpose I/O line 43 (does not belong to any 8-bit port).
44- PL_IO_NUM_44:	General-purpose I/O line 44 (does not belong to any 8-bit port).
45- PL_IO_NUM_45_CO ⁽³⁾ :	General-purpose I/O line 45 (does not belong to any 8-bit port). This line is also used by the beep ^[387] object to generate square wave output that is primarily intended for driving beeper (buzzer).
46- PL_IO_NUM_46:	General-purpose I/O line 46 (does not belong to any 8-bit port).
47- PL_IO_NUM_47:	General-purpose I/O line 47 (does not belong to any 8-bit port).

	8-bit port).
48- PL_IO_NUM_48:	General-purpose I/O line 48 (does not belong to any 8-bit port).
49- PL_IO_NUM_49:	General-purpose I/O line 49 (does not belong to any 8-bit port).
50- PL_IO_NUM_50:	General-purpose I/O line 50 (does not belong to any 8-bit port).
51- PL_IO_NUM_51:	General-purpose I/O line 51 (does not belong to any 8-bit port).
52- PL_IO_NUM_52:	General-purpose I/O line 52 (does not belong to any 8-bit port).
53- PL_IO_NUM_53:	General-purpose I/O line 53 (does not belong to any 8-bit port).
PL_IO_NULL:	This is a NULL line that does not physically exist. The state of this line is always detected as LOW. Setting this line has no effect.

Notes:

1. When a serial port is in the [UART](#) mode ([ser.mode](#)= 0- [PL_SER_MODE_UART](#)) this line is automatically configured to be an input when this serial port is enabled ([ser.enabled](#)= 1- YES) and returns to the previous input/output and high/low state when this serial port is closed ([ser.enabled](#)= 0- NO). When a serial port is in the [Wiegand](#) or [clock/data](#) mode ([ser.mode](#)= 1- [PL_SER_MODE_WIEGAND](#) or [ser.mode](#)= 2- [PL_SER_MODE_CLOCKDATA](#)), the line has to be configured as input by the application- this will not happen automatically.
2. When a serial port is in the UART mode ([ser.mode](#)= 0- [PL_SER_MODE_UART](#)) this line is automatically configured to be an output when the serial port is enabled ([ser.enabled](#)= 1- YES) and returns to the previous input/output and high/low state when the serial port is closed ([ser.enabled](#)= 0- NO). When a serial port is in the Wiegand or clock/data mode ([ser.mode](#)= 1- [PL_SER_MODE_WIEGAND](#) or [ser.mode](#)= 2- [PL_SER_MODE_CLOCKDATA](#)), the line has to be configured as output by the application- this will not happen automatically.
3. When the beeper pattern starts playing, this line is configured as output automatically. When the beeper pattern stops playing, the output returns to the input/output and high/low state that it had before the pattern started playing.

Enum pl_int_num

Enum [pl_int_num](#) contains the list of constants that refer to *interrupt* I/O lines available on the EM1000(W). Interrupt lines are mapped onto [general-purpose I/O lines](#). Interrupt line 0 corresponds to I/O line 16, interrupt line 1- to I/O lines 17, and so on. Keep in mind that for an interrupt line to work you need to configure a corresponding I/O line as input. Do this through the [io.num](#) property of the [io](#) object.

Enum [pl_int_num](#) includes the following constants:

PL_INT_NUM_0:	Interrupt line 0 (mapped onto I/O line 16).
PL_INT_NUM_1:	Interrupt line 1 (mapped onto I/O line 17).

PL_INT_NUM_2:	Interrupt line 2 (mapped onto I/O line 18).
PL_INT_NUM_3:	Interrupt line 3 (mapped onto I/O line 19).
PL_INT_NUM_4:	Interrupt line 4 (mapped onto I/O line 20).
PL_INT_NUM_5:	Interrupt line 5 (mapped onto I/O line 21).
PL_INT_NUM_6:	Interrupt line 6 (mapped onto I/O line 22).
PL_INT_NUM_7:	Interrupt line 7 (mapped onto I/O line 23).
PL_INT_NULL:	This is a NULL interrupt line that does not physically exist.

Enum pl_sock_interfaces

Enum pl_sock_interfaces contains the list of network interfaces supported by the EM1000(W). The EM1000 and EM1000W platforms differ in that the EM1000W has a W-Fi interface, which the EM1000 does not, and the pl_sock_interfaces reflects this.

EM1000 platform

0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET (default):	Ethernet interface.

EM1000W platform

0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET (default):	Ethernet interface.
2- PL_SOCK_INTERFACE_WLN:	Wi-Fi interface.

Platform-dependent Programming Information

This section contains miscellaneous information pertaining to the EM1000(W). Various objects described in the [Object Reference](#)^[212] direct you to this topic in all cases when object capabilities or behavior depends on the platform and, hence, cannot be described in the object reference itself. Each platform section in this manual has its own "Platform-dependent Programming Information" topic. If you are reading documentation top-to-bottom (we have never actually met anyone who does) you can skip this section now.

You have to explicitly configure I/O lines as inputs or outputs

On the EM1000(W) you need to explicitly enable or disable the output driver of each I/O line (controlled by the [io](#)^[365] object). The [io.enabled](#)^[370] property allows you to do this. When the driver is enabled (ser.enabled= 1-YES) and you read the state of the pin you will get back the state of your own output buffer. To turn the line into an input switch the output buffer off (ser.enabled= 0- NO). This will allow

you to sense the state of the external signal applied to the I/O line:

```
...
io.num= PL_IO_NUM_4      'just to select some line as an example
io.enabled= NO            'now the output driver is off
x=io.state                'read line state into x
...
```

When the device boots up all pins are configured as inputs. If you want to use any particular I/O pin as an output you need to enable the output driver first:

```
...
io.num= PL_IO_NUM_5      'select the line
io.enabled= YES           'enable output driver (you need to do this only
                           once)
io.state= LOW             'set the state
...
```



Make sure that your external circuitry does not attempt to drive the I/O lines that have their output buffers enabled. Severe damage to the device and/or your circuitry may occur if this happens!

You can remap RTS/W0out/cout and CTS/W0&1in/cin lines of the serial port

Two lines of the [serial port](#)^[224] -- **RTS/W0out/cout output**, and **CTS/W0&1in/cin input** -- can be reassigned (remapped) to other I/O pins of the device. This is done through [ser.rtsmap](#)^[261] and [ser.ctsmmap](#)^[249] properties.

By default, the mapping of these two lines is different for each serial port. See [Enum pl io num](#)^[142] for details.

Positions of **TX/W1out/dout output** and **RX/W1in/din input** are fixed and cannot be changed.

You have to explicitly configure certain I/O lines of the serial port as inputs or outputs

On the EM1000(W), it is necessary to configure some lines of the [serial port](#)^[224] as inputs our outputs. Depending on the mode of the serial port (see [ser.mode](#)^[255]) you need to set the following:

ser.mode	TX/ W1out/ dout output	RX/W1in/ din input	RTS/ W0out/ cout output	CTS/ W0&1in/ cin input
----------	---------------------------------	-----------------------	----------------------------------	------------------------------

<u>0-</u> <u>PL_SER_MODE_UAR</u> T ^[226]	Will auto-configure as output ⁽¹⁾	Will auto-configure as input ⁽¹⁾	Requires configuration as output ⁽²⁾	Requires configuration as input ⁽²⁾
<u>1-</u> <u>PL_SER_MODE_WIE</u> GAND ^[229]	Requires configuration as output	Requires configuration as input		
<u>2-</u> <u>PL_SER_MODE_CLO</u> CKDATA ^[232]				

Notes:

1. When This line does not require configuration, it will be configured automatically as input or output when the port is opened. When the port is closed the line will return to the input/output and high/low state it had before the port was opened.
2. Please, remember that you need to configure the I/O pin to which this line of the serial port is currently mapped.

Each serial port has 16 bytes of send FIFO

When the serial port^[224] is in the UART/full-duplex/flow control^[226] mode (ser.mode^[255]= 0- PL_SER_MODE_UART, ser.interface^[255]= 0- PL_SER_SI_FULLDUPLEX, and ser.flowcontrol^[253]= 1- ENABLED) the device is monitoring its CTS input to see if attached serial device is ready to receive more data. If the CTS state changes to "cannot transmit" the device will stop sending out data immediately. However, the data that has already entered the FIFO will still be sent out. Therefore, after the CTS state becomes "cannot transmit" the device can still send out up to 16 characters.

There is a PLL

On the EM1000(W) there is a PLL that, when switched on, increases the main clock of the device 8-fold (power cosumption also increases roughly by as much). When the PLL is off, the clock frequency of the device is 11.0592MHz; when the PLL is on the clock frequency is 88.4736MHz.

The clock frequency affects all aspects of device operation that rely on this clock. Naturally, program execution speed depends on the clock frequency. Additionally, the baudrates of the serial port^[224] (defined by the ser.baudrate^[248] property) depend on the main clock. Finally, the frequency of the square wave generated by the beep^[387] object depends on the main clock as well.

To deal with PLL, the sys^[212] object has a sys.currentpll^[218] read-only property and sys.newpll^[219] method. See PLL Management^[215] topic- it explains how to switch PLL on and off.

After the external reset (see sys.resettype^[222]) the EM1000 boots with the PLL on or off depending on the state of the PE pin.

For the serial port, there is a way to set the baudrate in the clock-independent (and, actually, platform-independent) way -- see ser.div9600^[250] property for details (example of use can be found in the Serial Settings^[224] topic). For the beep object, you just have to set the beep.divider^[388] correctly depending on the value returned by the sys.currpll property.

Data in the special configuration section of the EEPROM

Bottom 8 bytes of the EEPROM (accessible through the stor^[380] object) are reserved

for storing a MAC address of the device. On power-up, the MAC address is read out from the EEPROM and programmed into the Ethernet controller. You can always check current MAC through the [net.mac](#)^[269] read-only property of the [net](#)^[267] object but there is no direct way to change it. Instead, you can change the MAC address data in the EEPROM. Then, next time the device boots up it will start using this new address.

By default, the area storing MAC address is not accessible to your application- the [stor.base](#)^[381] property takes care of that. Unless you change it, this property specifies that your application's storage area starts at address 9 (counting from 1). To change MAC, set the stor.base to 1.

MAC address data in the EEPROM has a certain formatting -- you have to follow it if you want the MAC to be recognized by the firmware (TiOS). Here is the format:

Byte1	Byte2	Byte3	Byte4	Byte5	Byte 6	Byte7	Byte8
6	MAC0	MAC1	MAC2	MAC3	MAC4	MAC5	Checksum

Byte at address 1 must be set to 6- this means, that 6 byte of data follow (MAC address consists of 6 bytes). Addresses from 2 to 7 carry actual MAC data. Address 8 stores the checksum, which is calculated like this:

255-(modulo8_sum_of_addr_1_through_7)

Here is a sample code that stores new MAC address into the EEPROM and then reboots the device to make the device load this new MAC:

```

dim s as string
dim x as byte
...
s= "0.2.123.124.220.240" 'supposing, we want to set this MAC
...
...
s=chr(6)+ddval(s)      'added first byte (always 6) and converted
readable MAC into bytes
x=255-strsum(s)        'calculated checksum and assigned the
result to a BYTE variable (!!!)
s=s+chr(x)              'now our string is ready

stor.base(1)             'will access EEPROM from the bottom
x=stor.set(s,1)          'save data
if x<>len(s) then       'it is a good programming practice to check the
result
    'failed
else
    sys.reboot           'new MAC set, reboot!
end if

...

```

There are limitations on what MAC you can set. When loading the MAC into the Ethernet controller the device always resets the first byte of this address to 0. For example, if you set the MAC to 1.2.3.4.5.6 then the actual MAC used by the device will be 0.2.3.4.5.6.



If you write incorrect MAC data (wrong first byte or error in checksum calculation) the device will ignore it and boot up with default MAC, which is 0.1.2.3.4.100.

Available network interfaces

The EM1000 platform only has one network interface -- the Ethernet port. The [sock.allowedinterfaces](#)^[327] property refers to this interface as "NET". The EM1000W platform supports two network interfaces -- the Ethernet interface and Wi-Fi interface ("NET" and "WLN").

Also see the [pl_sock_interfaces](#)^[145] enum which is used by [sock.targetinterface](#)^[359] and [sock.currentinterface](#)^[331] properties.

Miscellaneous

- [Sys](#)^[212] object supports the [sys.onsystimerperiod](#)^[220] property and the [on sys timer](#)^[220] event generation period depends on the value of this property.

EM1202 and EM1202W Platforms

The difference between the EM1202 and EM1202W platforms is that the EM1202W additionally includes the [wln](#)^[497] (Wi-Fi) object, which works with an external GA1000 add-on module to function. All other features of these two platforms are exactly the same. Both platforms will be collectively referred to as "EM1202(W)".

- [Memory Space](#)^[149]
- [Supported Variable Types](#)^[183] (common for T1000-based devices)
- [LED signals](#)^[184] (common for all devices)
- [Debug communications](#)^[185] (common for all devices)
- [Project Settings Dialog](#)^[186] (common for all devices)
- [Supported Functions](#)^[183] (common for T1000-based devices)
- [Supported Objects](#)^[150]
- [Platform-dependent Constants](#)^[150]
- [Platform-dependent Programming Information](#)^[154]

Memory Space

The EM1202(W) has the following amounts of program (FLASH) and variable (RAM) memory available:

Program memory: 983040 Bytes

Variable memory: 20,480 bytes

EEPROM memory: 2048 bytes, of which 8 bytes are occupied by MAC address of the device*

*See [Platform-dependent programming information](#)^[154] for details.

Supported Objects

The following objects are found on this platform:

- [Sock](#)^[274] — socket communications (up to 16 UDP, TCP, and HTTP sessions);
- [Net](#)^[267] — controls Ethernet port;
- [WIn](#)^[497] — handles Wi-Fi interface (only available on the **EM1202W platform**, requires GA1000 add-on module);
- [Ser](#)^[224] — in charge of serial ports (UART, Wiegand, and clock/data modes);
- [Io](#)^[365] — handles I/O lines, ports, and interrupts;
- [Lcd](#)^[392] — controls graphical display panels (several types supported);
- [Kp](#)^[484] — scans keypads of matrix and "binary" types;
- [Fd](#)^[433] — manages flash memory file system and direct sector access;
- [Stor](#)^[380] — provides access to the EEPROM;
- [Romfile](#)^[375] — facilitates access to resource files (fixed data);
- [Pat](#)^[384] — "plays" patterns on up to five LED pairs;
- [Beep](#)^[387] — generates buzzer patterns;
- [Button](#)^[272] — monitors MD line (setup button);
- [Sys](#)^[212] — in charge of general device functionality.

Platform-dependent Constants

The following constant lists are platform-specific:

- [Enum pl_redir](#)^[147] - a list of constants that define buffer redirection (shorting) for this platform.
- [Enum pl_io_num](#)^[142] - a list of constants that define available I/O lines.
- [Enum pl_int_num](#)^[144] - a list of constants that define available *interrupt* lines.
- [Enum pl_sock_interfaces](#)^[154] - a list of available network interfaces.

Enum pl_redir

Enum pl_redir contains the list of constants that define buffer redirection (shorting) for this platform. The following objects support buffers and buffer redirection on the current platform:

- [Ser](#)^[224] object (see [ser.redir](#)^[260] method)
- [Sock](#)^[274] object (see [sock.redir](#)^[346] method)

Enum pl_redir for this platform includes the following constants:

- | | |
|-------------------|---|
| 0- PL_REDIR_NONE: | Cancels redirection for the serial port or socket. |
| 1- PL_REDIR_SER: | Redirects RX data of the serial port or socket to the TX buffer of the serial port 0. This constant can be used as a "base" for all other serial ports, i.e. in expressions like ser.redir= PL_REDIR_SER+f. |
| 1- PL_REDIR_SER0: | Redirects RX data of the serial port or socket to the TX buffer of the serial port 0. |
| 2- PL_REDIR_SER1: | Redirects RX data of the serial port or socket to the TX buffer of the serial port 1. |
| 3- PL_REDIR_SER: | Redirects RX data of the serial port or socket to the TX |

	buffer of the serial port 2.
4- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 3.
6- PL_REDIR_SOCK0:	Redirects RX data of the serial port or socket to the TX buffer of socket 0. This constant can be used as a "base" for all other sockets i.e. in expressions like <code>sock.redirect= PL_REDIR_SOCK0+f</code> .
7- PL_REDIR_SOCK1:	Redirects RX data of the serial port or socket to the TX buffer of socket 1.
8- PL_REDIR_SOCK2:	Redirects RX data of the serial port or socket to the TX buffer of socket 2.
9- PL_REDIR_SOCK3:	Redirects RX data of the serial port or socket to the TX buffer of socket 3.
10- PL_REDIR_SOCK4:	Redirects RX data of the serial port or socket to the TX buffer of socket 4.
11- PL_REDIR_SOCK5:	Redirects RX data of the serial port or socket to the TX buffer of socket 5.
12- PL_REDIR_SOCK6:	Redirects RX data of the serial port or socket to the TX buffer of socket 6.
13- PL_REDIR_SOCK7:	Redirects RX data of the serial port or socket to the TX buffer of socket 7.
14- PL_REDIR_SOCK8:	Redirects RX data of the serial port or socket to the TX buffer of socket 8.
15- PL_REDIR_SOCK9:	Redirects RX data of the serial port or socket to the TX buffer of socket 9.
16- PL_REDIR_SOCK10:	Redirects RX data of the serial port or socket to the TX buffer of socket 10.
17- PL_REDIR_SOCK11:	Redirects RX data of the serial port or socket to the TX buffer of socket 11.
18- PL_REDIR_SOCK12:	Redirects RX data of the serial port or socket to the TX buffer of socket 12.
19- PL_REDIR_SOCK13:	Redirects RX data of the serial port or socket to the TX buffer of socket 13.
20- PL_REDIR_SOCK14:	Redirects RX data of the serial port or socket to the TX buffer of socket 14.
21- PL_REDIR_SOCK15:	Redirects RX data of the serial port or socket to the TX buffer of socket 15.

Enum pl_io_num

Enum `pl_io_num` contains the list of constants that refer to I/O lines available on this platform. Use these constants when selecting the line with the `io`^[365] object (see the `io.num`^[372] property).

All I/O lines of the EM1202(W) require explicit configuration as inputs or outputs -- this is done through the `io.enabled`^[370] property of the `io`^[365] object. On power-up, all lines are configured as inputs. When the line is configured for input its output driver is tri-stated. When the line is configured for output, its output driver is enabled. It is possible to read the state of the line even when it is working as an output.

Certain lines automatically become inputs and outputs in certain modes of

operation -- see below for details.



If you are using the EM1202 platform to work with the EM1202EV board or DS1202 controller, please keep in mind that these two products interconnect the I/O lines of the EM1202 in a certain way. Specifically, each line of the 8-bit port 1 is connected to a corresponding line from port 2. For example, line PL_IO_NUM_9_RX0 is connected to the line PL_IO_NUM17_INT1. Be careful to avoid configuring both lines in such pairs as outputs. This can permanently damage the EM1202. If the PL_IO_NUM_9_RX0 is an output, then the PL_IO_NUM17_INT1 should not be an output as well. Alternatively, just use a dedicated [DS1202 platform](#)^[166] to work with the EM1202EV or the DS1202. This platform defines the I/O lines differently and you have nothing to worry about there.

Enum pl_io_num for this platform includes the following constants:

PL_IO_NUM_0:	Selects general-purpose I/O line 0 (P0.0).
PL_IO_NUM_1:	Selects general-purpose I/O line 1 (P0.1).
PL_IO_NUM_2:	Selects general-purpose I/O line 2 (P0.2).
PL_IO_NUM_3:	Selects general-purpose I/O line 3 (P0.3).
PL_IO_NUM_4:	Selects general-purpose I/O line 4 (P0.4).
PL_IO_NUM_5:	Selects general-purpose I/O line 5 (P0.5).
PL_IO_NUM_6:	Selects general-purpose I/O line 6 (P0.6).
PL_IO_NUM_7:	Selects general-purpose I/O line 7 (P0.7).
PL_IO_NUM_8_RX0:	Selects general-purpose I/O line 8 (P1.0). This line is also the RX/W1in/din ^[225] input of the serial port 0.
PL_IO_NUM_9_TX0:	Selects general-purpose I/O line 9 (P1.1). This line is also the TX/W1out/dout ^[226] output of the serial port 0.
PL_IO_NUM_10_RX1:	Selects general-purpose I/O line 10 (P1.2). This line is also the RX/W0&1in/din input of the serial port 1.
PL_IO_NUM_11_TX1:	Selects general-purpose I/O line 11 (P1.3). This line is also the TX/W1out/dout output of the serial port 1.
PL_IO_NUM_12_RX2:	Selects general-purpose I/O line 12 (P1.4). This line is also the RX/W0&1in/din input of the serial port 2.
PL_IO_NUM_13_TX2:	Selects general-purpose I/O line 13 (P1.5). This line is also the TX/W1out/dout output of the serial port 2.
PL_IO_NUM_14_RX3:	Selects general-purpose I/O line 14 (P1.6). This line is also the RX/W0&1in/din input of the serial port 3.
PL_IO_NUM_15_TX3:	Selects general-purpose I/O line 15 (P1.7). This line is also the TX/W1out/dout output of the serial port 3.
PL_IO_NUM_16_INT0:	Selects general-purpose I/O line 16 (P2.0).
PL_IO_NUM_17_INT1:	Selects general-purpose I/O line 17 (P2.1).
PL_IO_NUM_18_INT2:	Selects general-purpose I/O line 18 (P2.2).
PL_IO_NUM_19_INT3:	Selects general-purpose I/O line 19 (P2.3).
PL_IO_NUM_20_INT4:	Selects general-purpose I/O line 20 (P2.4).
PL_IO_NUM_21_INT5:	Selects general-purpose I/O line 21 (P2.5).

PL_IO_NUM_22_INT6:	Selects general-purpose I/O line 22 (P2.6).
PL_IO_NUM_23_INT7:	Selects general-purpose I/O line 23 (P2.7).
PL_IO_NUM_24:	Selects general-purpose I/O line 24 (does not belong to any 8-bit port).
PL_IO_NUM_25:	Selects general-purpose I/O line 25 (does not belong to any 8-bit port).
PL_IO_NUM_26:	Selects general-purpose I/O line 26 (does not belong to any 8-bit port).
PL_IO_NUM_27:	Selects general-purpose I/O line 27 (does not belong to any 8-bit port).
PL_IO_NUM_28:	Selects general-purpose I/O line 28 (does not belong to any 8-bit port).
PL_IO_NUM_29_CO:	Selects general-purpose I/O line 29 (does not belong to any 8-bit port). This line is also used by the beep ^[38] object to generate square wave output that is primarily intended for driving beeper (buzzer).
PL_IO_NUM_30:	Selects general-purpose I/O line 30 (does not belong to any 8-bit port).
PL_IO_NUM_31:	Selects general-purpose I/O line 31 (does not belong to any 8-bit port).
PL_IO_NULL:	This is a NULL line that does not physically exist. The state of this line is always detected as LOW. Setting this line has no effect.

Notes:

- When a serial port is in the [UART](#)^[226] mode ([ser.mode](#)^[255]= 0- [PL_SER_MODE_UART](#)) this line is automatically configured to be an input when this serial port is enabled ([ser.enabled](#)^[251]= 1- YES) and returns to the previous input/output and high/low state when this serial port is closed ([ser.enabled](#)= 0- NO). When a serial port is in the [Wiegand](#)^[229] or [clock/data](#)^[232] mode ([ser.mode](#)= 1- [PL_SER_MODE_WIEGAND](#) or [ser.mode](#)= 2- [PL_SER_MODE_CLOCKDATA](#)), the line has to be configured as input by the application- this will not happen automatically.
- When a serial port is in the UART mode ([ser.mode](#)= 0- [PL_SER_MODE_UART](#)) this line is automatically configured to be an output when the serial port is enabled ([ser.enabled](#)= 1- YES) and returns to the previous input/output and high/low state when the serial port is closed ([ser.enabled](#)= 0- NO). When a serial port is in the Wiegand or clock/data mode ([ser.mode](#)= 1- [PL_SER_MODE_WIEGAND](#) or [ser.mode](#)= 2- [PL_SER_MODE_CLOCKDATA](#)), the line has to be configured as output by the application- this will not happen automatically.
- When the beeper pattern starts playing, this line is configured as output automatically. When the beeper pattern stops playing, the output returns to the input/output and high/low state that it had before the pattern started playing.

Enum pl_int_num

Enum [pl_int_num](#) contains the list of constants that refer to *interrupt* I/O lines available on the EM1202(W). Interrupt lines are mapped onto [general-purpose I/O lines](#)^[142]. Interrupt line 0 corresponds to I/O line 16, interrupt line 1- to I/O lines

17, and so on. Keep in mind that for an interrupt line to work you need to configure a corresponding I/O line as input. Do this through the [io.num](#)^[372] property of the [io](#)^[365] object.

Enum pl_int_num for this platform includes the following constants:

PL_INT_NUM_0:	Interrupt line 0 (mapped onto I/O line 16).
PL_INT_NUM_1:	Interrupt line 1 (mapped onto I/O line 17).
PL_INT_NUM_2:	Interrupt line 2 (mapped onto I/O line 18).
PL_INT_NUM_3:	Interrupt line 3 (mapped onto I/O line 19).
PL_INT_NUM_4:	Interrupt line 4 (mapped onto I/O line 20).
PL_INT_NUM_5:	Interrupt line 5 (mapped onto I/O line 21).
PL_INT_NUM_6:	Interrupt line 6 (mapped onto I/O line 22).
PL_INT_NUM_7:	Interrupt line 7 (mapped onto I/O line 23).
PL_INT_NULL:	This is a NULL interrupt line that does not physically exist.

Enum pl_sock_interfaces

Enum pl_sock_interfaces contains the list of network interfaces supported by the EM1202(W). The EM1202 and EM1202W platforms differ in that the EM1202W has a W-Fi interface, which the EM1202 does not, and the pl_sock_interfaces reflects this.

EM1202 platform

0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET (default):	Ethernet interface.

EM1202W platform

0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET (default):	Ethernet interface.
2- PL_SOCK_INTERFACE_WLN:	Wi-Fi interface.

Platform-dependent Programming Information

This section contains miscellaneous information that sets the EM1202(W) apart from other platforms. Various objects described in the [Object Reference](#)^[212] direct you to this topic in all cases when object capabilities or behavior depends on the platform and, hence, cannot be described in the object reference itself. Each platform section in this manual has its own "Platform-dependent Programming Information" topic. If you are reading documentation top-to-bottom (we have never actually met anyone who does) you can skip this section now.

You have to explicitly configure I/O lines as inputs or outputs

On the EM1202(W) you need to explicitly enable or disable the output driver of each I/O line (controlled by the [io](#)^[365] object). The [io.enabled](#)^[370] property allows you to do this. When the driver is enabled (ser.enabled= 1-YES) and you read the state of the pin you will get back the state of your own output buffer. To turn the line into an input switch the output buffer off (ser.enabled= 0- NO). This will allow you to sense the state of the external signal applied to the I/O line:

```
...
io.num= PL_IO_NUM_4      'just to select some line as an example
io.enabled= NO            'now the output driver is off
x=io.state                'read line state into x
...
...
```

When the device boots up all pins are configured as inputs. If you want to use any particular I/O pin as an output you need to enable the output driver first:

```
...
io.num= PL_IO_NUM_5      'select the line
io.enabled= YES           'enable output driver (you need to do this only
                           once)
io.state= LOW             'set the state
...
...
```



Make sure that your external circuitry does not attempt to drive the I/O lines that have their output buffers enabled. Severe damage to the device and/or your circuitry may occur if this happens!

You can remap RTS/W0out/cout and CTS/W0&1in/cin lines of the serial port

Two lines of the [serial port](#)^[224] -- **RTS/W0out/cout output**, and **CTS/W0&1in/cin input** -- can be reassigned (remapped) to other I/O pins of the device. This is done through [ser.rtsmap](#)^[261] and [ser.ctsmmap](#)^[249] properties.

By default, the mapping of these two lines is different for each serial port. See [Enum pl_io_num](#)^[151] for details.

Positions of **TX/W1out/dout output** and **RX/W1in/din input** are fixed and cannot be changed.

You have to explicitly configure certain I/O lines of the serial port as inputs or outputs

On the EM1202(W), it is necessary to configure the lines of the [serial port](#)^[224] as inputs our outputs. Depending on the mode of the serial port (see [ser.mode](#)^[255]) you need to set the following:

ser.mode	TX/ W1out/ dout output	RX/W1in/ din input	RTS/ W0out/ cout output	CTS/ W0&1in/ cin input
0- PL SER MODE_UAR T^[226]	Will auto-configure as output ⁽¹⁾	Will auto-configure as input ⁽¹⁾	Requires configuration as output ⁽²⁾	Requires configuration as input ⁽²⁾
1- PL SER MODE_WIE GAND^[229]	Requires configuration as output	Requires configuration as input		
2- PL SER MODE_CLO CKDATA^[232]				

Notes:

1. When This line does not require configuration, it will be configured automatically as input or output when the port is opened. When the port is closed the line will return to the input/output and high/low state it had before the port was opened.
2. Please, remember that you need to configure the I/O pin to which this line of the serial port is currently mapped.

Each serial port has 16 bytes of send FIFO

When the [serial_port^{\[224\]}](#) is in the [UART/full-duplex/flow control^{\[226\]}](#) mode ([ser.mode^{\[255\]}](#)= 0- PL_SER_MODE_UART, [ser.interface^{\[255\]}](#)= 0- PL_SER_SI_FULLDUPLEX, and [ser.flowcontrol^{\[253\]}](#)= 1- ENABLED) the device is monitoring its CTS input to see if attached serial device is ready to receive more data. If the CTS state changes to "cannot transmit" the device will stop sending out data immediately. However, the data that has already entered the FIFO will still be sent out. Therefore, after the CTS state becomes "cannot transmit" the device can still send out up to 16 characters.

There is a PLL

On the EM1202(W) there is a PLL that, when switched on, increases the main clock of the device 8-fold (power cosumption also increases roughly by as much). When the PLL is off, the clock frequency of the device is 11.0592MHz; when the PLL is on the clock frequency is 88.4736MHz.

There is no hardware input to select default power-on state of the PLL. After the external reset (see [sys.resettype^{\[222\]}](#)) the EM1202 boots with the PLL ON.

For PLL control, the [sys^{\[212\]}](#) object has a [sys.currentpll^{\[218\]}](#) read-only property and [sys.newpll^{\[219\]}](#) method. See [PLL Management^{\[215\]}](#) topic- it explains how to switch PLL on and off.

The clock frequency affects all aspects of device operation that rely on this clock. Naturally, program execution speed depends on this clock. Additionally, the baudrates of the [serial_port^{\[224\]}](#) (defined by the [ser.baudrate^{\[248\]}](#) property) depend on the main clock. Finally, the frequency of the square wave generated by the [beep^{\[387\]}](#) object depends on the main clock as well.

For the serial port, there is a way to set the baudrate in the clock-independent (and, actually, platform-independent) way -- see [ser.div9600^{\[250\]}](#) property for details, (example of use can be found in the [Serial Settings^{\[224\]}](#) topic). For the beep object, you just have to set the [beep.divider^{\[388\]}](#) correctly depending on the value

returned by the sys.currpl property.

Data in the special configuration section of the EEPROM

Bottom 8 bytes of the EEPROM (accessible through the [stor](#)^[38] object) are reserved for storing MAC address of the device. On power-up, the MAC address is read out from the EEPROM and programmed into the Ethernet controller. You can always check current MAC through the [net.mac](#)^[269] read-only property of the [net](#)^[267] object but there is no direct way to change it. Instead, you can change the MAC address data in the EEPROM. Then, next time the device boots up it will start using this new address.

By default, the area storing MAC address is not accessible to your application- the [stor.base](#)^[38] property takes care of that. Unless you change it, this property specifies that your application's storage area starts at address 9 (counting from 1). To change MAC, set the stor.base to 1.

MAC address data in the EEPROM has a certain formatting -- you have to follow it if you want the MAC to be recognized by the firmware (TiOS). Here is the format:

Byte1	Byte2	Byte3	Byte4	Byte5	Byte 6	Byte7	Byte8
6	MAC0	MAC1	MAC2	MAC3	MAC 4	MAC5	Checksum

Byte at address 1 must be set to 6- this means, that 6 byte of data follow (MAC address consists of 6 bytes). Addresses from 2 to 7 carry actual MAC data. Address 8 stores the checksum, which is calculated like this:

$$255 - (\text{modulo8_sum_of_addr_1_through_7})$$

Here is a sample code that stores new MAC address into the EEPROM and then reboots the device to make the device load this new MAC:

```

dim s as string
dim x as byte
...
s= "0.2.123.124.220.240" 'supposing, we want to set this MAC
...
...
s=chr(6)+ddval(s)      'added first byte (always 6) and converted
readable MAC into bytes
x=255-strsum(s)        'calculated checksum and assigned the
result to a BYTE variable (!!!)
s=s+chr(x)              'now our string is ready

stor.base(1)            'will access EEPROM from the bottom
x=stor.set(s,1)          'save data
if x<>len(s) then       'it is a good programming practice to check the
result
    'failed
else
    sys.reboot           'new MAC set, reboot!
end if

...

```

There are limitations on what MAC you can set. When loading the MAC into the Ethernet controller the device always resets the first byte of this address to 0. For

example, if you set the MAC to 1.2.3.4.5.6 then the actual MAC used by the device will be 0.2.3.4.5.6.



If you write incorrect MAC data (wrong first byte or error in checksum calculation) the device will ignore it and boot up with default MAC, which is 0.1.2.3.4.100.

Available network interfaces

The EM1206 platform only has one network interface -- the Ethernet port. The [sock.allowedinterfaces](#)^[327] property refers to this interface as "NET". The EM1206W platform supports two network interfaces -- the Ethernet interface and Wi-Fi interface ("NET" and "WLN").

Also see the [pl_sock_interfaces](#)^[145] enum which is used by [sock.targetinterface](#)^[359] and [sock.currentinterface](#)^[331] properties.

Miscellaneous

- [Sys](#)^[212] object supports the [sys.onsystimerperiod](#)^[220] property and the [on sys timer](#)^[220] event generation period depends on the value of this property.

EM1206 and EM1206W Platforms

The difference between the EM1206 and EM1206W platforms is that the EM1206W additionally includes the [wln](#)^[497] (Wi-Fi) object, which works with an external GA1000 add-on module to function. All other features of these two platforms are exactly the same. Both platforms will be collectively referred to as "EM1206(W)".

- [Memory Space](#)^[158]
- [Supported Variable Types](#)^[183] (common for T1000-based devices)
- [LED signals](#)^[184] (common for all devices)
- [Debug communications](#)^[185] (common for all devices)
- [Project Settings Dialog](#)^[186] (common for all devices)
- [Supported Functions](#)^[183] (common for T1000-based devices)
- [Supported Objects](#)^[159]
- [Platform-dependent Constants](#)^[159]
- [Platform-dependent Programming Information](#)^[163]

Memory Space

The EM1206(W) has the following amounts of program (FLASH) and variable (RAM) memory available:

Program memory: 458,752 Bytes for the EM1206(W)-512K;
983040 Bytes for the EM1206(W)-1024K

Variable memory: 20,480 bytes

EEPROM memory: 2048 bytes, of which 8 bytes are occupied by MAC address of the device*

*See [Platform-dependent programming information](#)^[163] for details.

Supported Objects

The following objects are found on the EM1206(W):

- [Sock](#)^[274] — socket communications (up to 16 UDP, TCP, and HTTP sessions);
- [Net](#)^[267] — controls Ethernet port;
- [WIn](#)^[497] — handles Wi-Fi interface (only available on the **EM1206W platform**, requires GA1000 add-on module);
- [Ser](#)^[224] — in charge of serial ports (UART, Wiegand, and clock/data modes);
- [Io](#)^[365] — handles I/O lines, ports, and interrupts;
- [Kp](#)^[484] — scans keypads of matrix and "binary" types;
- [Rtc](#)^[389] — keeps track of date and time;
- [Fd](#)^[433] — manages flash memory file system and direct sector access;
- [Stor](#)^[380] — provides access to the EEPROM;
- [Romfile](#)^[375] — facilitates access to resource files (fixed data);
- [Pat](#)^[384] — "plays" patterns on up to five LED pairs;
- [Beep](#)^[387] — generates buzzer patterns;
- [Button](#)^[272] — monitors MD line (setup button);
- [Sys](#)^[212] — in charge of general device functionality.

Platform-dependent Constants

The following constant lists are platform-specific:

- [Enum pl_redir](#)^[159]- a list of constants that define buffer redirection (shorting) for this platform.
- [Enum pl_io_num](#)^[160]- a list of constants that define available I/O lines.
- [Enum pl_int_num](#)^[162]- a list of constants that define available *interrupt* lines.
- [Enum pl_sock_interfaces](#)^[162]- a list of available network interfaces.

Enum pl_redir

Enum pl_redir contains the list of constants that define buffer redirection (shorting) for this platform. The following objects support buffers and buffer redirection on the EM1206(W):

- [Ser](#)^[224] object (see [ser.redir](#)^[260] method)
- [Sock](#)^[274] object (see [sock.redir](#)^[346] method)

Enum pl_redir for this platform includes the following constants:

- | | |
|-------------------|---|
| 0- PL_REDIR_NONE: | Cancels redirection for the serial port or socket. |
| 1- PL_REDIR_SER: | Redirects RX data of the serial port or socket to the TX buffer of the serial port 0. This constant can be used as a "base" for all other serial ports, i.e. in expressions like ser.redir= PL_REDIR_SER+f. |
| 1- PL_REDIR_SER0: | Redirects RX data of the serial port or socket to the TX buffer of the serial port 0. |
| 2- PL_REDIR_SER1: | Redirects RX data of the serial port or socket to the TX buffer of the serial port 1. |
| 3- PL_REDIR_SER: | Redirects RX data of the serial port or socket to the TX |

	buffer of the serial port 2.
4- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 3.
6- PL_REDIR SOCK0:	Redirects RX data of the serial port or socket to the TX buffer of socket 0. This constant can be used as a "base" for all other sockets i.e. in expressions like <code>sock.redirect= PL_REDIR SOCK0+f</code> .
7- PL_REDIR SOCK1:	Redirects RX data of the serial port or socket to the TX buffer of socket 1.
8- PL_REDIR SOCK2:	Redirects RX data of the serial port or socket to the TX buffer of socket 2.
9- PL_REDIR SOCK3:	Redirects RX data of the serial port or socket to the TX buffer of socket 3.
10- PL_REDIR SOCK4:	Redirects RX data of the serial port or socket to the TX buffer of socket 4.
11- PL_REDIR SOCK5:	Redirects RX data of the serial port or socket to the TX buffer of socket 5.
12- PL_REDIR SOCK6:	Redirects RX data of the serial port or socket to the TX buffer of socket 6.
13- PL_REDIR SOCK7:	Redirects RX data of the serial port or socket to the TX buffer of socket 7.
14- PL_REDIR SOCK8:	Redirects RX data of the serial port or socket to the TX buffer of socket 8.
15- PL_REDIR SOCK9:	Redirects RX data of the serial port or socket to the TX buffer of socket 9.
16- PL_REDIR SOCK10:	Redirects RX data of the serial port or socket to the TX buffer of socket 10.
17- PL_REDIR SOCK11:	Redirects RX data of the serial port or socket to the TX buffer of socket 11.
18- PL_REDIR SOCK12:	Redirects RX data of the serial port or socket to the TX buffer of socket 12.
19- PL_REDIR SOCK13:	Redirects RX data of the serial port or socket to the TX buffer of socket 13.
20- PL_REDIR SOCK14:	Redirects RX data of the serial port or socket to the TX buffer of socket 14.
21- PL_REDIR SOCK15:	Redirects RX data of the serial port or socket to the TX buffer of socket 15.

Enum pl_io_num

Enum `pl_io_num` contains the list of constants that refer to I/O lines available on this platform. Use these constants when selecting the line with the `io`^[365] object (see the `io.num`^[372] property).

All I/O lines of the EM1206(W) require explicit configuration as inputs or outputs -- this is done through the `io.enabled`^[370] property of the `io`^[365] object. On power-up, all lines are configured as inputs. When the line is configured for input its output driver is tri-stated. When the line is configured for output, its output driver is enabled. It is possible to read the state of the line even when it is working as an output.

Certain lines automatically become inputs and outputs in certain modes of

operation -- see below for details.

Enum pl_io_num includes the following constants:

- 0- PL_IO_NUM_0_RX0_INT0⁽¹⁾: General-purpose I/O line 0 (P0.0). This line is also the [RX/W1in/din](#)^[225] input of the serial port 0 and the interrupt line 0.
- 1- PL_IO_NUM_1_TX0_INT1⁽²⁾: General-purpose I/O line 1 (P0.1). This line is also the [TX/W1out/dout](#)^[225] output of the serial port 0 and the interrupt line 1.
- 2- PL_IO_NUM_2_RX1_INT2⁽¹⁾: General-purpose I/O line 2 (P0.2). This line is also the RX/W0&1in/din input of the serial port 1 and the interrupt line 2.
- 3- PL_IO_NUM_3_TX1_INT3⁽²⁾: General-purpose I/O line 3 (P0.3). This line is also the TX/W1out/dout output of the serial port 1 and the interrupt line 3.
- 4- PL_IO_NUM_4_RX2_INT4⁽¹⁾: General-purpose I/O line 4 (P0.4). This line is also the RX/W0&1in/din input of the serial port 2 and the interrupt line 4.
- 5- PL_IO_NUM_5_TX2_INT5⁽²⁾: General-purpose I/O line 5 (P0.5). This line is also the TX/W1out/dout output of the serial port 2 and the interrupt line 5.
- 6- PL_IO_NUM_6_RX3_INT6⁽¹⁾: General-purpose I/O line 6 (P0.6). This line is also the RX/W0&1in/din input of the serial port 3 and the interrupt line 6.
- 7- PL_IO_NUM_7_TX3_INT7⁽²⁾: General-purpose I/O line 7 (P0.7). This line is also the TX/W1out/dout output of the serial port 3 and the interrupt line 7.
- 8- PL_IO_NUM_8: General-purpose I/O line 8 (P1.0).
- 9- PL_IO_NUM_9: General-purpose I/O line 9 (P1.1).
- 10- PL_IO_NUM_10: General-purpose I/O line 10 (P1.2).
- 11- PL_IO_NUM_11: General-purpose I/O line 11 (P1.3).
- 12- PL_IO_NUM_12: General-purpose I/O line 12 (P1.4).
- 13- PL_IO_NUM_13: General-purpose I/O line 13 (P1.5).
- 14- PL_IO_NUM_14: General-purpose I/O line 14 (P1.6).
- 15- PL_IO_NUM_15: General-purpose I/O line 15 (P1.7).
- 16- PL_IO_NUM_16_CO⁽³⁾: General-purpose I/O line 16 (does not belong to any 8-bit port). This line is also used by the [beep](#)^[387] object to generate square wave output that is primarily intended for driving beeper (buzzer).
- PL_IO_NULL: This is a NULL line that does not physically exist. The state of this line is always detected as LOW. Setting this line has no effect.

Notes:

1. When a serial port is in the [UART](#)^[226] mode ([ser.mode](#)^[255]= 0-[PL_SER_MODE_UART](#)) this line is automatically configured to be an input when this serial port is enabled ([ser.enabled](#)^[254]= 1- YES) and returns to the previous input/output and high/low state when this serial port is closed ([ser.enabled](#)= 0-

- NO). When a serial port is in the [Wiegand](#)^[229] or [clock/data](#)^[232] mode (ser.mode= 1- PL_SER_MODE_WIEGAND or ser.mode= 2- PL_SER_MODE_CLOCKDATA), the line has to be configured as input by the application- this will not happen automatically.
2. When a serial port is in the UART mode (ser.mode= 0- PL_SER_MODE_UART) this line is automatically configured to be an output when the serial port is enabled (ser.enabled= 1- YES) and returns to the previous input/output and high/low state when the serial port is closed (ser.enabled= 0- NO). When a serial port is in the Wiegand or clock/data mode (ser.mode= 1- PL_SER_MODE_WIEGAND or ser.mode= 2- PL_SER_MODE_CLOCKDATA), the line has to be configured as output by the application- this will not happen automatically.
 3. When the beeper pattern stars playing, this line is configured as output automatically. When the beeper pattern stops playing, the output returns to the input/output and high/low state that it had before the pattern started playing.

Enum pl_int_num

Enum pl_int_num contains the list of constants that refer to *interrupt I/O lines* available on the EM1206(W). Interrupt lines are mapped onto [general-purpose I/O lines](#)^[142]. Interrupt line 0 corresponds to I/O line 0, interrupt line 1- to I/O lines 1, and so on. Keep in mind that for an interrupt line to work you need to configure a corresponding I/O line as input. Do this through the [io.num](#)^[372] property of the [io](#)^[365] object.

Enum pl_int_num includes the following constants:

PL_INT_NUM_0:	Interrupt line 0 (mapped onto I/O line 0).
PL_INT_NUM_1:	Interrupt line 1 (mapped onto I/O line 1).
PL_INT_NUM_2:	Interrupt line 2 (mapped onto I/O line 2).
PL_INT_NUM_3:	Interrupt line 3 (mapped onto I/O line 3).
PL_INT_NUM_4:	Interrupt line 4 (mapped onto I/O line 4).
PL_INT_NUM_5:	Interrupt line 5 (mapped onto I/O line 5).
PL_INT_NUM_6:	Interrupt line 6 (mapped onto I/O line 6).
PL_INT_NUM_7:	Interrupt line 7 (mapped onto I/O line 7).
PL_INT_NULL:	This is a NULL interrupt line that does not physically exist.

Enum pl_sock_interfaces

Enum pl_sock_interfaces contains the list of network interfaces supported by the EM1206(W). The EM1206 and EM1206W platforms differ in that the EM1206W has a W-Fi interface, which the EM1206 does not, and the pl_sock_interfaces reflects this.

EM1206 platform

0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET (default):	Ethernet interface.

EM1206W platform

0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET (default):	Ethernet interface.
2- PL_SOCK_INTERFACE_WLN:	Wi-Fi interface.

Platform-dependent Programming Information

This section contains miscellaneous information pertaining to the EM1206(W). Various objects described in the [Object Reference](#)^[21] direct you to this topic in all cases when object capabilities or behavior depends on the platform and, hence, cannot be described in the object reference itself. Each platform section in this manual has its own "Platform-dependent Programming Information" topic. If you are reading documentation top-to-bottom (we have never actually met anyone who does) you can skip this section now.

You have to explicitly configure I/O lines as inputs or outputs

On the EM1206(W) you need to explicitly enable or disable the output driver of each I/O line (controlled by the [io](#)^[365] object). The [io.enabled](#)^[370] property allows you to do this. When the driver is enabled (ser.enabled= 1-YES) and you read the state of the pin you will get back the state of your own output buffer. To turn the line into an input switch the output buffer off (ser.enabled= 0- NO). This will allow you to sense the state of the external signal applied to the I/O line:

```
...
io.num= PL_IO_NUM_4      'just to select some line as an example
io.enabled= NO            'now the output driver is off
x=io.state                'read line state into x
...
```

When the device boots up all pins are configured as inputs. If you want to use any particular I/O pin as an output you need to enable the output driver first:

```
...
io.num= PL_IO_NUM_5      'select the line
io.enabled= YES           'enable output driver (you need to do this only
                           once)
io.state= LOW              'set the state
...
```



Make sure that your external circuitry does not attempt to drive the I/O lines that have their output buffers enabled. Severe damage to the device and/or your circuitry may occur if this happens!

You can remap RTS/W0out/cout and CTS/W0&1in/cin lines of the serial port

Two lines of the [serial port](#)^[224] -- **RTS/W0out/cout output**, and **CTS/W0&1in/cin input** -- can be reassigned (remapped) to other I/O pins of the device. This is done through [ser.rtsmap](#)^[261] and [ser.ctsmmap](#)^[249] properties.

By default, the mapping of these two lines is different for each serial port. See [Enum pl_io_num](#)^[142] for details.

Positions of **TX/W1out/dout output** and **RX/W1in/din input** are fixed and cannot be changed.

You have to explicitly configure certain I/O lines of the serial port as inputs or outputs

On the EM1206(W), it is necessary to configure some lines of the [serial port](#)^[224] as inputs our outputs. Depending on the mode of the serial port (see [ser.mode](#)^[255]) you need to set the following:

ser.mode	TX/ W1out/ dout output	RX/W1in/ din input	RTS/ W0out/ cout output	CTS/ W0&1in/ cin input
0- PL_SER_MODE_UAR T ^[226]	Will auto-configure as output ⁽¹⁾	Will auto-configure as input ⁽¹⁾	Requires configuration as output ⁽²⁾	Requires configuration as input ⁽²⁾
1- PL_SER_MODE_WIE GAND ^[229]	Requires configuration as output	Requires configuration as input		
2- PL_SER_MODE_CLO CKDATA ^[232]				

Notes:

1. When This line does not require configuration, it will be configured automatically as input or output when the port is opened. When the port is closed the line will return to the input/output and high/low state it had before the port was opened.
2. Please, remember that you need to configure the I/O pin to which this line of the serial port is currently mapped.

Each serial port has 16 bytes of send FIFO

When the [serial port](#)^[224] is in the [UART/full-duplex/flow control](#)^[226] mode ([ser.mode](#)^[255]= 0- [PL_SER_MODE_UART](#), [ser.interface](#)^[255]= 0- [PL_SER_SI_FULLDUPLEX](#), and [ser.flowcontrol](#)^[253]= 1- [ENABLED](#)) the device is monitoring its CTS input to see if attached serial device is ready to receive more data. If the CTS state changes to "cannot transmit" the device will stop sending out data immediately. However, the data that has already entered the FIFO will still be sent out. Therefore, after the CTS state becomes "cannot transmit" the device can still send out up to 16 characters.

There is a PLL

On the EM1206(W) there is a PLL that, when switched on, increases the main clock of the device 8-fold (power consumption also increases roughly by as much). When the PLL is off, the clock frequency of the device is 11.0592MHz; when the PLL is on the clock frequency is 88.4736MHz.

The clock frequency affects all aspects of device operation that rely on this clock. Naturally, program execution speed depends on the clock frequency. Additionally, the baudrates of the [serial port](#)^[224] (defined by the [ser.baudrate](#)^[248] property) depend on the main clock. Finally, the frequency of the square wave generated by the [beep](#)^[387] object depends on the main clock as well.

To deal with PLL, the [sys](#)^[212] object has a [sys.currentpll](#)^[218] read-only property and [sys.newpll](#)^[219] method. See [PLL Management](#)^[215] topic- it explains how to switch PLL on and off.

After the external reset (see [sys.resettype](#)^[222]) the EM1206 boots with the PLL on or off depending on the state of the PE pin.

For the serial port, there is a way to set the baudrate in the clock-independent (and, actually, platform-independent) way -- see [ser.div9600](#)^[250] property for details (example of use can be found in the [Serial Settings](#)^[224] topic). For the beep object, you just have to set the [beep.divider](#)^[388] correctly depending on the value returned by the sys.currpll property.

Data in the special configuration section of the EEPROM

Bottom 8 bytes of the EEPROM (accessible through the [stor](#)^[380] object) are reserved for storing a MAC address of the device. On power-up, the MAC address is read out from the EEPROM and programmed into the Ethernet controller. You can always check current MAC through the [net.mac](#)^[269] read-only property of the [net](#)^[267] object but there is no direct way to change it. Instead, you can change the MAC address data in the EEPROM. Then, next time the device boots up it will start using this new address.

By default, the area storing MAC address is not accessible to your application- the [stor.base](#)^[381] property takes care of that. Unless you change it, this property specifies that your application's storage area starts at address 9 (counting from 1). To change MAC, set the stor.base to 1.

MAC address data in the EEPROM has a certain formatting -- you have to follow it if you want the MAC to be recognized by the firmware (TiOS). Here is the format:

Byte1	Byte2	Byte3	Byte4	Byte5	Byte 6	Byte7	Byte8
6	MAC0	MAC1	MAC2	MAC3	MAC 4	MAC5	Checksum

Byte at address 1 must be set to 6- this means, that 6 byte of data follow (MAC address consists of 6 bytes). Addresses from 2 to 7 carry actual MAC data. Address 8 stores the checksum, which is calculated like this:

$$255 - (\text{modulo8_sum_of_addr_1_through_7})$$

Here is a sample code that stores new MAC address into the EEPROM and then reboots the device to make the device load this new MAC:

```

dim s as string
dim x as byte
...
s= "0.2.123.124.220.240" 'supposing, we want to set this MAC
...
...
s=chr(6)+ddval(s)      'added first byte (always 6) and converted
readable MAC into bytes
x=255-strsum(s)        'calculated checksum and assigned the
result to a BYTE variable (!!!)
s=s+chr(x)              'now our string is ready

stor.base(1)             'will access EEPROM from the bottom
x=stor.set(s,1)          'save data
if x<>len(s) then       'it is a good programming practice to check the
result
    'failed
else
    sys.reboot           'new MAC set, reboot!
end if

...

```

There are limitations on what MAC you can set. When loading the MAC into the Ethernet controller the device always resets the first byte of this address to 0. For example, if you set the MAC to 1.2.3.4.5.6 then the actual MAC used by the device will be 0.2.3.4.5.6.



If you write incorrect MAC data (wrong first byte or error in checksum calculation) the device will ignore it and boot up with default MAC, which is 0.1.2.3.4.100.

Available network interfaces

The EM1206 platform only has one network interface -- the Ethernet port. The [sock.allowedinterfaces](#)^[327] property refers to this interface as "NET". The EM1206W platform supports two network interfaces -- the Ethernet interface and Wi-Fi interface ("NET" and "WLN").

Also see the [pl_sock_interfaces](#)^[145] enum which is used by [sock.targetinterface](#)^[359] and [sock.currentinterface](#)^[331] properties.

Miscellaneous

- [Sys](#)^[212] object supports the [sys.onsystimerperiod](#)^[220] property and the [on sys timer](#)^[220] event generation period depends on the value of this property.

DS1202 Platform

- [Memory Space](#)^[167]
- [Supported Variable Types](#)^[183] (common for T1000-based devices)
- [LED signals](#)^[184] (common for all devices)
- [Debug communications](#)^[185] (common for all devices)
- [Project Settings Dialog](#)^[186] (common for all devices)
- [Supported Functions](#)^[183] (common for T1000-based devices)
- [Supported Objects](#)^[167]

- [Platform-dependent Constants](#)^[167]
- [Platform-dependent Programming Information](#)^[171]

Memory Space

The platform has the following amounts of program (FLASH) and variable (RAM) memory available:

Program memory:	983040 Bytes
Variable memory:	20,480 bytes
EEPROM memory:	2048 bytes, of which 8 bytes are occupied by MAC address of the device*

*See [Platform-dependent programming information](#)^[171] for details.

Supported Objects

The following objects are found on this platform :

- [Sock](#)^[274] — socket communications (up to 16 UDP, TCP, and HTTP sessions);
- [Net](#)^[267] — controls Ethernet port;
- [Ser](#)^[224] — in charge of serial ports (UART, Wiegand, and clock/data modes);
- [Io](#)^[365] — handles I/O lines, ports, and interrupts;
- [Fd](#)^[433] — manages flash memory file system and direct sector access;
- [Stor](#)^[380] — provides access to the EEPROM;
- [Romfile](#)^[375] — facilitates access to resource files (fixed data);
- [Pat](#)^[384] — "plays" patterns on up to five LED pairs;
- [Button](#)^[272] — monitors MD line (setup button);
- [Sys](#)^[212] — in charge of general device functionality.

Platform-dependent Constants

The following constant lists are platform-specific:

- [Enum pl_redir](#)^[167]- a list of constants that define buffer redirection (shorting) for this platform.
- [Enum pl_io_num](#)^[169]- a list of constants that define available I/O lines.
- [Enum pl_int_num](#)^[170]- a list of constants that define available *interrupt* lines.
- [Enum pl_sock_interfaces](#)^[170]- a list of available network interfaces.

Enum pl_redir

Enum pl_redir contains the list of constants that define buffer redirection (shorting) for this platform. The following objects support buffers and buffer redirection on this platform:

- [Ser](#)^[224] object (see [ser.redir](#)^[260] method)
- [Sock](#)^[274] object (see [sock.redir](#)^[346] method)

Enum pl_redir for this platform includes the following constants:

- | | |
|-------------------|--|
| 0- PL_REDIR_NONE: | Cancels redirection for the serial port or socket. |
|-------------------|--|

1- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0. This constant can be used as a "base" for all other serial ports, i.e. in expressions like ser.redirect= PL_REDIR_SER+f.
1- PL_REDIR_SER0:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0.
2- PL_REDIR_SER1:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 1.
3- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 2.
4- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 3.
6- PL_REDIR_SOCK0:	Redirects RX data of the serial port or socket to the TX buffer of socket 0. This constant can be used as a "base" for all other sockets i.e. in expressions like sock.redirect= PL_REDIR_SOCK0+f.
7- PL_REDIR_SOCK1:	Redirects RX data of the serial port or socket to the TX buffer of socket 1.
8- PL_REDIR_SOCK2:	Redirects RX data of the serial port or socket to the TX buffer of socket 2.
9- PL_REDIR_SOCK3:	Redirects RX data of the serial port or socket to the TX buffer of socket 3.
10- PL_REDIR_SOCK4:	Redirects RX data of the serial port or socket to the TX buffer of socket 4.
11- PL_REDIR_SOCK5:	Redirects RX data of the serial port or socket to the TX buffer of socket 5.
12- PL_REDIR_SOCK6:	Redirects RX data of the serial port or socket to the TX buffer of socket 6.
13- PL_REDIR_SOCK7:	Redirects RX data of the serial port or socket to the TX buffer of socket 7.
14- PL_REDIR_SOCK8:	Redirects RX data of the serial port or socket to the TX buffer of socket 8.
15- PL_REDIR_SOCK9:	Redirects RX data of the serial port or socket to the TX buffer of socket 9.
16- PL_REDIR_SOCK10:	Redirects RX data of the serial port or socket to the TX buffer of socket 10.
17- PL_REDIR_SOCK11:	Redirects RX data of the serial port or socket to the TX buffer of socket 11.
18- PL_REDIR_SOCK12:	Redirects RX data of the serial port or socket to the TX buffer of socket 12.
19- PL_REDIR_SOCK13:	Redirects RX data of the serial port or socket to the TX buffer of socket 13.
20- PL_REDIR_SOCK14:	Redirects RX data of the serial port or socket to the TX buffer of socket 14.
21- PL_REDIR_SOCK15:	Redirects RX data of the serial port or socket to the TX buffer of socket 15.

Enum pl_io_num

Enum pl_io_num contains the list of constants that refer to I/O lines available on this platform. Use these constants when selecting the line with the [io](#)^[365] object (see the [io.num](#)^[372] property).

All I/O lines of the DS1202 platform require explicit configuration as inputs or outputs -- this is done through the [io.enabled](#)^[370] property of the [io](#)^[365] object. On power-up, all lines are configured as inputs. When the line is configured for input its output driver is tri-stated. When the line is configured for output, its output driver is enabled. It is possible to read the state of the line even when it is working as an output.



The EM1202EV-RS board and the DS1202 controller have RS232 transceiver IC onboard. This IC defines which I/O lines of the device should be configured as inputs, and which- as outputs. Specifically, do not try to use I/I lines 0, 2, 4, 6, and 7 as outputs -- this can permanently damage the hardware.

Certain lines automatically become inputs and outputs in certain modes of operation -- see below for details.

Enum pl_io_num includes the following constants:

- 0- PL_IO_NUM_0_RX0_INT0⁽¹⁾: General-purpose I/O line 0 (P0.0). This line is also the [RX/W1in/din](#)^[225] input of the serial port 0 and the interrupt line 0.
- 1- PL_IO_NUM_1_TX0_INT4⁽²⁾: General-purpose I/O line 1 (P0.1). This line is also the [TX/W1out/dout](#)^[225] output of the serial port 0 and the interrupt line 4.
- 2- PL_IO_NUM_2_RX1_INT1⁽¹⁾: General-purpose I/O line 2 (P0.2). This line is also the RX/W0&1in/din input of the serial port 1 and the interrupt line 1.
- 3- PL_IO_NUM_3_TX1_INT5⁽²⁾: General-purpose I/O line 3 (P0.3). This line is also the TX/W1out/dout output of the serial port 1 and the interrupt line 5.
- 4- PL_IO_NUM_4_RX2_INT2⁽¹⁾: General-purpose I/O line 4 (P0.4). This line is also the RX/W0&1in/din input of the serial port 2 and the interrupt line 2.
- 5- PL_IO_NUM_5_TX2_INT6⁽²⁾: General-purpose I/O line 5 (P0.5). This line is also the TX/W1out/dout output of the serial port 2 and the interrupt line 6.
- 6- PL_IO_NUM_6_RX3_INT3⁽¹⁾: General-purpose I/O line 6 (P0.6). This line is also the RX/W0&1in/din input of the serial port 3 and the interrupt line 3.
- 7- PL_IO_NUM_7_INT7⁽²⁾: General-purpose I/O line 7 (P0.7). This line is also the interrupt line 7.
- PL_IO_NULL: This is a NULL line that does not physically exist. The state of this line is always detected as LOW. Setting this line has no effect.

Notes:

1. **Should be configured as an input.** When a serial port is in the [UART](#)^[226] mode (`ser.mode`^[25]= 0- `PL_SER_MODE_UART`) this line is automatically configured to be an input when this serial port is enabled (`ser.enabled`^[25]= 1- YES) and returns to the previous input/output and high/low state when this serial port is closed (`ser.enabled`= 0- NO). When a serial port is in the [Wiegand](#)^[229] or [clock/data](#)^[232] mode (`ser.mode`= 1- `PL_SER_MODE_WIEGAND` or `ser.mode`= 2- `PL_SER_MODE_CLOCKDATA`), the line has to be configured as input by the application- this will not happen automatically.
2. **Should be configured as an output.** When a serial port is in the [UART](#) mode (`ser.mode`= 0- `PL_SER_MODE_UART`) this line is automatically configured to be an output when the serial port is enabled (`ser.enabled`= 1- YES) and returns to the previous input/output and high/low state when the serial port is closed (`ser.enabled`= 0- NO). When a serial port is in the [Wiegand](#) or [clock/data](#) mode (`ser.mode`= 1- `PL_SER_MODE_WIEGAND` or `ser.mode`= 2- `PL_SER_MODE_CLOCKDATA`), the line has to be configured as output by the application- this will not happen automatically.

Enum pl_int_num

Enum `pl_int_num` contains the list of constants that refer to *interrupt I/O lines* available on this platform. Interrupt lines are mapped onto [general-purpose I/O lines](#)^[142]. Keep in mind that for an interrupt line to work you need to configure a corresponding I/O line as input. Do this through the `io.num`^[372] property of the `io`^[365] object.

Enum `pl_int_num` includes the following constants:

<code>PL_INT_NUM_0:</code>	Interrupt line 0 (mapped onto I/O line 0).
<code>PL_INT_NUM_1:</code>	Interrupt line 1 (mapped onto I/O line 2).
<code>PL_INT_NUM_2:</code>	Interrupt line 2 (mapped onto I/O line 4).
<code>PL_INT_NUM_3:</code>	Interrupt line 3 (mapped onto I/O line 6).
<code>PL_INT_NUM_4:</code>	Interrupt line 4 (mapped onto I/O line 1).
<code>PL_INT_NUM_5:</code>	Interrupt line 5 (mapped onto I/O line 3).
<code>PL_INT_NUM_6:</code>	Interrupt line 6 (mapped onto I/O line 5).
<code>PL_INT_NUM_7:</code>	Interrupt line 7 (mapped onto I/O line 7).
<code>PL_INT_NULL:</code>	This is a NULL interrupt line that does not physically exist.

Enum pl_sock_interfaces

Enum `pl_sock_interfaces` contains the list of network interfaces supported by the platform:

<code>0- PL_SOCK_INTERFACE_NULL:</code>	Null (empty) interface.
<code>1- PL_SOCK_INTERFACE_NET (default):</code>	Ethernet interface.

Platform-dependent Programming Information

This section contains miscellaneous information pertaining to the DS1202 platform. Various objects described in the [Object Reference](#)^[212] direct you to this topic in all cases when object capabilities or behavior depends on the platform and, hence, cannot be described in the object reference itself. Each platform section in this manual has its own "Platform-dependent Programming Information" topic. If you are reading documentation top-to-bottom (we have never actually met anyone who does) you can skip this section now.

You have to explicitly configure I/O lines as inputs or outputs

On the DS1202 platform you need to explicitly enable or disable the output driver of each I/O line (controlled by the [io](#)^[365] object). The [io.enabled](#)^[370] property allows you to do this. When the driver is enabled (ser.enabled= 1-YES) and you read the state of the pin you will get back the state of your own output buffer. To turn the line into an input switch the output buffer off (ser.enabled= 0- NO). This will allow you to sense the state of the external signal applied to the I/O line:

```
...
io.num= PL_IO_NUM_4      'just to select some line as an example
io.enabled= NO            'now the output driver is off
x=io.state                'read line state into x
...
...
```

When the device boots up all pins are configured as inputs. If you want to use any particular I/O pin as an output you need to enable the output driver first:

```
...
io.num= PL_IO_NUM_5      'select the line
io.enabled= YES           'enable output driver (you need to do this only
                           once)
io.state= LOW             'set the state
...
...
```



Make sure that your external circuitry does not attempt to drive the I/O lines that have their output buffers enabled. Severe damage to the device and/or your circuitry may occur if this happens!

You can remap RTS/W0out/cout and CTS/W0&1in/cin lines of the serial port

Two lines of the [serial port](#)^[224] -- **RTS/W0out/cout output**, and **CTS/W0&1in/ cin input** -- can be reassigned (remapped) to other I/O pins of the device. This is done through [ser.rtsmap](#)^[261] and [ser.ctsmap](#)^[249] properties.

By default, the mapping of these two lines is different for each serial port. See [Enum pl io num](#)^[142] for details.

Positions of **TX/W1out/dout output** and **RX/W1in/din input** are fixed and cannot be changed.

You have to explicitly configure certain I/O lines of the serial port as

inputs or outputs

On this platform it is necessary to configure some lines of the [serial port](#)^[224] as inputs our outputs. Depending on the mode of the serial port (see [ser.mode](#)^[255]) you need to set the following:

ser.mode	TX/ W1out/ dout output	RX/W1in/ din input	RTS/ W0out/ cout output	CTS/ W0&1in/ cin input
0- PL SER MODE_UAR T ^[226]	Will auto-configure as output ⁽¹⁾	Will auto-configure as input ⁽¹⁾	Requires configuration as output ⁽²⁾	Requires configuration as input ⁽²⁾
1- PL SER MODE_WIE GAND ^[229]	Requires configuration as output	Requires configuration as input		
2- PL SER MODE_CLO CKDATA ^[232]				

Notes:

1. When This line does not require configuration, it will be configured automatically as input or output when the port is opened. When the port is closed the line will return to the input/output and high/low state it had before the port was opened.
2. Please, remember that you need to configure the I/O pin to which this line of the serial port is currently mapped.

Each serial port has 16 bytes of send FIFO

When the [serial port](#)^[224] is in the [UART/full-duplex/flow control](#)^[226] mode ([ser.mode](#)^[255]= 0- PL_SER_MODE_UART, [ser.interface](#)^[255]= 0- PL_SER_SI_FULLDUPLEX, and [ser.flowcontrol](#)^[253]= 1- ENABLED) the device is monitoring its CTS input to see if attached serial device is ready to receive more data. If the CTS state changes to "cannot transmit" the device will stop sending out data immediately. However, the data that has already entered the FIFO will still be sent out. Therefore, after the CTS state becomes "cannot transmit" the device can still send out up to 16 characters.

There is a PLL

On the DS1202 platform there is a PLL that, when switched on, increases the main clock of the device 8-fold (power cosumption also increases roughly by as much). When the PLL is off, the clock frequency of the device is 11.0592MHz; when the PLL is on the clock frequency is 88.4736MHz.

The clock frequency affects all aspects of device operation that rely on this clock. Naturally, program execution speed depends on the clock frequency. Additionally, the baudrates of the [serial port](#)^[224] (defined by the [ser.baudrate](#)^[248] property) depend on the main clock. Finally, the frequency of the square wave generated by the [beep](#)^[387] object depends on the main clock as well.

To deal with PLL, the [sys](#)^[212] object has a [sys.currentpll](#)^[218] read-only property and [sys.newpll](#)^[219] method. See [PLL Management](#)^[215] topic- it explains how to switch PLL on and off.

After the external reset (see [sys.resettype](#)^[222]) the DS1202 boots with the PLL on or off depending on the state of the PE pin.

For the serial port, there is a way to set the baudrate in the clock-independent (and, actually, platform-independent) way -- see [ser.div9600](#)^[250] property for details (example of use can be found in the [Serial Settings](#)^[224] topic). For the beep object, you just have to set the [beep.divider](#)^[388] correctly depending on the value returned by the sys.currpll property.

Data in the special configuration section of the EEPROM

Bottom 8 bytes of the EEPROM (accessible through the [stor](#)^[380] object) are reserved for storing a MAC address of the device. On power-up, the MAC address is read out from the EEPROM and programmed into the Ethernet controller. You can always check current MAC through the [net.mac](#)^[269] read-only property of the [net](#)^[267] object but there is no direct way to change it. Instead, you can change the MAC address data in the EEPROM. Then, next time the device boots up it will start using this new address.

By default, the area storing MAC address is not accessible to your application- the [stor.base](#)^[381] property takes care of that. Unless you change it, this property specifies that your application's storage area starts at address 9 (counting from 1). To change MAC, set the stor.base to 1.

MAC address data in the EEPROM has a certain formatting -- you have to follow it if you want the MAC to be recognized by the firmware (TiOS). Here is the format:

Byte1	Byte2	Byte3	Byte4	Byte5	Byte 6	Byte7	Byte8
6	MAC0	MAC1	MAC2	MAC3	MAC 4	MAC5	Checksum

Byte at address 1 must be set to 6- this means, that 6 byte of data follow (MAC address consists of 6 bytes). Addresses from 2 to 7 carry actual MAC data. Address 8 stores the checksum, which is calculated like this:

$255 - (\text{modulo8_sum_of_addr_1_through_7})$

Here is a sample code that stores new MAC address into the EEPROM and then reboots the device to make the device load this new MAC:

```

dim s as string
dim x as byte
...
s= "0.2.123.124.220.240" 'supposing, we want to set this MAC
...
...
s=chr(6)+ddval(s)      'added first byte (always 6) and converted
readable MAC into bytes
x=255-strsum(s)        'calculated checksum and assigned the
result to a BYTE variable (!!!)
s=s+chr(x)              'now our string is ready

stor.base(1)             'will access EEPROM from the bottom
x=stor.set(s,1)          'save data
if x<>len(s) then       'it is a good programming practice to check the
result
    'failed
else
    sys.reboot           'new MAC set, reboot!
end if

...

```

There are limitations on what MAC you can set. When loading the MAC into the Ethernet controller the device always resets the first byte of this address to 0. For example, if you set the MAC to 1.2.3.4.5.6 then the actual MAC used by the device will be 0.2.3.4.5.6.



If you write incorrect MAC data (wrong first byte or error in checksum calculation) the device will ignore it and boot up with default MAC, which is 0.1.2.3.4.100.

Available network interfaces

The DS1202 platform only has one network interface -- the Ethernet port. The [sock.allowedinterfaces](#)^[327] property refers to this interface as "NET".

Also see the [pl_sock_interfaces](#)^[145] enum which is used by [sock.targetinterface](#)^[359] and [sock.currentinterface](#)^[331] properties.

Miscellaneous

- [Sys](#)^[212] object supports the [sys.onsystemtimerperiod](#)^[220] property and the [on sys timer](#)^[220] event generation period depends on the value of this property.

DS1206 Platform

- [Memory Space](#)^[175]
- [Supported Variable Types](#)^[183] (common for T1000-based devices)
- [LED signals](#)^[184] (common for all devices)
- [Debug communications](#)^[185] (common for all devices)
- [Project Settings Dialog](#)^[186] (common for all devices)
- [Supported Functions](#)^[183] (common for T1000-based devices)
- [Supported Objects](#)^[175]
- [Platform-dependent Constants](#)^[175]
- [Platform-dependent Programming Information](#)^[179]

Memory Space

The platform has the following amounts of program (FLASH) and variable (RAM) memory available:

Program memory: 458,752 Bytes for the DS1206-512K; 983040 Bytes for the DS1206-1024K

Variable memory: 20,480 bytes

EEPROM memory: 2048 bytes, of which 8 bytes are occupied by MAC address of the device*

*See [Platform-dependent programming information](#)^[179] for details.

Supported Objects

The following objects are found on this platform :

- [Sock](#)^[274] — socket communications (up to 16 UDP, TCP, and HTTP sessions);
- [Net](#)^[267] — controls Ethernet port;
- [Ser](#)^[224] — in charge of serial ports (UART, Wiegand, and clock/data modes);
- [Io](#)^[365] — handles I/O lines, ports, and interrupts;
- [Fd](#)^[433] — manages flash memory file system and direct sector access;
- [Stor](#)^[380] — provides access to the EEPROM;
- [Romfile](#)^[375] — facilitates access to resource files (fixed data);
- [Pat](#)^[384] — "plays" patterns on up to five LED pairs;
- [Button](#)^[272] — monitors MD line (setup button);
- [Sys](#)^[212] — in charge of general device functionality.

Platform-dependent Constants

The following constant lists are platform-specific:

- [Enum pl_redir](#)^[175]- a list of constants that define buffer redirection (shorting) for this platform.
- [Enum pl_io_num](#)^[177]- a list of constants that define available I/O lines.
- [Enum pl_int_num](#)^[178]- a list of constants that define available *interrupt* lines.
- [Enum pl_sock_interfaces](#)^[179]- a list of available network interfaces.

Enum pl_redir

Enum pl_redir contains the list of constants that define buffer redirection (shorting) for this platform. The following objects support buffers and buffer redirection on this platform:

- [Ser](#)^[224] object (see [ser.redir](#)^[260] method)
- [Sock](#)^[274] object (see [sock.redir](#)^[346] method)

Enum pl_redir for this platform includes the following constants:

- | | |
|-------------------|--|
| 0- PL_REDIR_NONE: | Cancels redirection for the serial port or socket. |
| 1- PL_REDIR_SER: | Redirects RX data of the serial port or socket to the TX buffer of the serial port 0. This constant can be used as |

	a "base" for all other serial ports, i.e. in expressions like ser.redir= PL_REDIR_SER+f.
1- PL_REDIR_SER0:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0.
2- PL_REDIR_SER1:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 1.
3- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 2.
4- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 3.
6- PL_REDIR_SOCK0:	Redirects RX data of the serial port or socket to the TX buffer of socket 0. This constant can be used as a "base" for all other sockets i.e. in expressions like sock.redir= PL_REDIR_SOCK0+f.
7- PL_REDIR_SOCK1:	Redirects RX data of the serial port or socket to the TX buffer of socket 1.
8- PL_REDIR_SOCK2:	Redirects RX data of the serial port or socket to the TX buffer of socket 2.
9- PL_REDIR_SOCK3:	Redirects RX data of the serial port or socket to the TX buffer of socket 3.
10- PL_REDIR_SOCK4:	Redirects RX data of the serial port or socket to the TX buffer of socket 4.
11- PL_REDIR_SOCK5:	Redirects RX data of the serial port or socket to the TX buffer of socket 5.
12- PL_REDIR_SOCK6:	Redirects RX data of the serial port or socket to the TX buffer of socket 6.
13- PL_REDIR_SOCK7:	Redirects RX data of the serial port or socket to the TX buffer of socket 7.
14- PL_REDIR_SOCK8:	Redirects RX data of the serial port or socket to the TX buffer of socket 8.
15- PL_REDIR_SOCK9:	Redirects RX data of the serial port or socket to the TX buffer of socket 9.
16- PL_REDIR_SOCK10:	Redirects RX data of the serial port or socket to the TX buffer of socket 10.
17- PL_REDIR_SOCK11:	Redirects RX data of the serial port or socket to the TX buffer of socket 11.
18- PL_REDIR_SOCK12:	Redirects RX data of the serial port or socket to the TX buffer of socket 12.
19- PL_REDIR_SOCK13:	Redirects RX data of the serial port or socket to the TX buffer of socket 13.
20- PL_REDIR_SOCK14:	Redirects RX data of the serial port or socket to the TX buffer of socket 14.
21- PL_REDIR_SOCK15:	Redirects RX data of the serial port or socket to the TX buffer of socket 15.

Enum pl_io_num

Enum pl_io_num contains the list of constants that refer to I/O lines available on this platform. Use these constants when selecting the line with the [io](#)^[365] object (see the [io.num](#)^[372] property).

All I/O lines of the DS1206 platform require explicit configuration as inputs or outputs -- this is done through the [io.enabled](#)^[370] property of the [io](#)^[365] object. On power-up, all lines are configured as inputs. When the line is configured for input its output driver is tri-stated. When the line is configured for output, its output driver is enabled. It is possible to read the state of the line even when it is working as an output.



The DS1206N-RS board and the DS1206 controller have RS232 transceiver IC onboard. This IC defines which I/O lines of the device should be configured as inputs, and which- as outputs. Specifically, do not try to use I/I lines 0, 2, 4, and 6 as outputs -- this can permanently damage the hardware.

Certain lines automatically become inputs and outputs in certain modes of operation -- see below for details.

Enum pl_io_num includes the following constants:

- 0- PL_IO_NUM_0_RX0_INT0⁽¹⁾: General-purpose I/O line 0 (P0.0). This line is also the [RX/W1in/din](#)^[225] input of the serial port 0 and the interrupt line 0.
- 1- PL_IO_NUM_1_TX0_INT1⁽²⁾: General-purpose I/O line 1 (P0.1). This line is also the [TX/W1out/dout](#)^[225] output of the serial port 0 and the interrupt line 1.
- 2- PL_IO_NUM_2_RX1_INT2⁽¹⁾: General-purpose I/O line 2 (P0.2). This line is also the RX/W0&1in/din input of the serial port 1 and the interrupt line 2.
- 3- PL_IO_NUM_3_TX1_INT3⁽²⁾: General-purpose I/O line 3 (P0.3). This line is also the TX/W1out/dout output of the serial port 1 and the interrupt line 3.
- 4- PL_IO_NUM_4_RX2_INT4⁽¹⁾: General-purpose I/O line 4 (P0.4). This line is also the RX/W0&1in/din input of the serial port 2 and the interrupt line 4.
- 5- PL_IO_NUM_5_TX2_INT5⁽²⁾: General-purpose I/O line 5 (P0.5). This line is also the TX/W1out/dout output of the serial port 2 and the interrupt line 5.
- 6- PL_IO_NUM_6_RX3_INT6⁽¹⁾: General-purpose I/O line 6 (P0.6). This line is also the RX/W0&1in/din input of the serial port 3 and the interrupt line 6.
- 7- PL_IO_NUM_7_EMPTY: General-purpose I/O line 7 (P0.7). This line is also the interrupt line 7.
- 8- PL_IO_NUM_8_PWRROUT: Controls the power output on pin 9 of the DB9M connector (this applies only to DS1206N-RS and DS1206 devices). Power will be ON when this output is enabled (io.enabled= 1- YES) and set to HIGH (io.state= 1- HIGH).
- PL_IO_NULL: This is a NULL line that does not physically exist.

The state of this line is always detected as LOW.
Setting this line has no effect.

Notes:

1. **Should be configured as an input.** When a serial port is in the [UART](#)^[226] mode (`ser.mode`^[255]= 0- `PL_SER_MODE_UART`) this line is automatically configured to be an input when this serial port is enabled (`ser.enabled`^[251]= 1- YES) and returns to the previous input/output and high/low state when this serial port is closed (`ser.enabled`= 0- NO). When a serial port is in the [Wiegand](#)^[229] or [clock/data](#)^[232] mode (`ser.mode`= 1- `PL_SER_MODE_WIEGAND` or `ser.mode`= 2- `PL_SER_MODE_CLOCKDATA`), the line has to be configured as input by the application- this will not happen automatically.
2. **Should be configured as an output.** When a serial port is in the [UART](#) mode (`ser.mode`= 0- `PL_SER_MODE_UART`) this line is automatically configured to be an output when the serial port is enabled (`ser.enabled`= 1- YES) and returns to the previous input/output and high/low state when the serial port is closed (`ser.enabled`= 0- NO). When a serial port is in the [Wiegand](#) or [clock/data](#) mode (`ser.mode`= 1- `PL_SER_MODE_WIEGAND` or `ser.mode`= 2- `PL_SER_MODE_CLOCKDATA`), the line has to be configured as output by the application- this will not happen automatically.

Enum pl_int_num

Enum `pl_int_num` contains the list of constants that refer to *interrupt I/O lines available on this platform*. Interrupt lines are mapped onto [general-purpose I/O lines](#)^[142]. Keep in mind that for an interrupt line to work you need to configure a corresponding I/O line as input. Do this through the `io.num`^[372] property of the `io`^[365] object.

Enum `pl_int_num` includes the following constants:

<code>PL_INT_NUM_0:</code>	Interrupt line 0 (mapped onto I/O line 0).
<code>PL_INT_NUM_1:</code>	Interrupt line 1 (mapped onto I/O line 1).
<code>PL_INT_NUM_2:</code>	Interrupt line 2 (mapped onto I/O line 2).
<code>PL_INT_NUM_3:</code>	Interrupt line 3 (mapped onto I/O line 3).
<code>PL_INT_NUM_4:</code>	Interrupt line 4 (mapped onto I/O line 4).
<code>PL_INT_NUM_5:</code>	Interrupt line 5 (mapped onto I/O line 5).
<code>PL_INT_NUM_6:</code>	Interrupt line 6 (mapped onto I/O line 6).
<code>PL_INT_NUM_7:</code>	Interrupt line 7 (mapped onto I/O line 7).
<code>PL_INT_NULL:</code>	This is a NULL interrupt line that does not physically exist.

Enum pl_sock_interfaces

Enum pl_sock_interfaces contains the list of network interfaces supported by the platform:

0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET (default):	Ethernet interface.

Platform-dependent Programming Information

This section contains miscellaneous information pertaining to the DS1206 platform. Various objects described in the [Object Reference](#)^[212] direct you to this topic in all cases when object capabilities or behavior depends on the platform and, hence, cannot be described in the object reference itself. Each platform section in this manual has its own "Platform-dependent Programming Information" topic. If you are reading documentation top-to-bottom (we have never actually met anyone who does) you can skip this section now.

You have to explicitly configure I/O lines as inputs or outputs

On the DS1206 platform you need to explicitly enable or disable the output driver of each I/O line (controlled by the [io](#)^[365] object). The [io.enabled](#)^[370] property allows you to do this. When the driver is enabled (ser.enabled= 1-YES) and you read the state of the pin you will get back the state of your own output buffer. To turn the line into an input switch the output buffer off (ser.enabled= 0- NO). This will allow you to sense the state of the external signal applied to the I/O line:

```
...
io.num= PL_IO_NUM_4      'just to select some line as an example
io.enabled= NO           'now the output driver is off
x=io.state               'read line state into x
...
```

When the device boots up all pins are configured as inputs. If you want to use any particular I/O pin as an output you need to enable the output driver first:

```
...
io.num= PL_IO_NUM_5      'select the line
io.enabled= YES          'enable output driver (you need to do this only
                         once)
io.state= LOW            'set the state
...
```



Make sure that your external circuitry does not attempt to drive the I/O lines that have their output buffers enabled. Severe damage to the device and/or your circuitry may occur if this happens!

You can remap RTS/W0out/cout and CTS/W0&1in/cin lines of the serial port

Two lines of the [serial port](#)^[224] -- **RTS/W0out/cout output**, and **CTS/W0&1in/cin input** -- can be reassigned (remapped) to other I/O pins of the device. This is done through [ser.rtsmap](#)^[261] and [ser.ctsmmap](#)^[249] properties.

By default, the mapping of these two lines is different for each serial port. See [Enum pl_io_num](#)^[142] for details.

Positions of **TX/W1out/dout output** and **RX/W1in/din input** are fixed and cannot be changed.

You have to explicitly configure certain I/O lines of the serial port as inputs or outputs

On this platform it is necessary to configure some lines of the [serial port](#)^[224] as inputs our outputs. Depending on the mode of the serial port (see [ser.mode](#)^[255]) you need to set the following:

ser.mode	TX/ W1out/ dout output	RX/W1in/ din input	RTS/ W0out/ cout output	CTS/ W0&1in/ cin input
0- PL_SER_MODE_UAR T ^[226]	Will auto-configure as output ⁽¹⁾	Will auto-configure as input ⁽¹⁾	Requires configuration as output ⁽²⁾	Requires configuration as input ⁽²⁾
1- PL_SER_MODE_WIE GAND ^[229]	Requires configuration as output	Requires configuration as input		
2- PL_SER_MODE_CLO CKDATA ^[232]				

Notes:

1. When This line does not require configuration, it will be configured automatically as input or output when the port is opened. When the port is closed the line will return to the input/output and high/low state it had before the port was opened.
2. Please, remember that you need to configure the I/O pin to which this line of the serial port is currently mapped.

Each serial port has 16 bytes of send FIFO

When the [serial port](#)^[224] is in the [UART/full-duplex/flow control](#)^[226] mode ([ser.mode](#)^[255]= 0- [PL_SER_MODE_UART](#), [ser.interface](#)^[255]= 0- [PL_SER_SI_FULLDUPLEX](#), and [ser.flowcontrol](#)^[253]= 1- [ENABLED](#)) the device is monitoring its CTS input to see if attached serial device is ready to receive more data. If the CTS state changes to "cannot transmit" the device will stop sending out data immediately. However, the data that has already entered the FIFO will still be sent out. Therefore, after the CTS state becomes "cannot transmit" the device can still send out up to 16 characters.

There is a PLL

On the DS1206 platform there is a PLL that, when switched on, increases the main clock of the device 8-fold (power cosumption also increases roughly by as much). When the PLL is off, the clock frequency of the device is 11.0592MHz; when the PLL is on the clock frequency is 88.4736MHz.

The clock frequency affects all aspects of device operation that rely on this clock. Naturally, program execution speed depends on the clock frequency. Additionally, the baudrates of the [serial port](#)^[224] (defined by the [ser.baudrate](#)^[248] property) depend on the main clock. Finally, the frequency of the square wave generated by the [beep](#)^[387] object depends on the main clock as well.

To deal with PLL, the [sys](#)^[212] object has a [sys.currentpll](#)^[218] read-only property and [sys.newpll](#)^[219] method. See [PLL Management](#)^[215] topic- it explains how to switch PLL on and off.

After the external reset (see [sys.resettype](#)^[222]) the DS1206 boots with the PLL on or off depending on the state of the PE pin.

For the serial port, there is a way to set the baudrate in the clock-independent (and, actually, platform-independent) way -- see [ser.div9600](#)^[250] property for details (example of use can be found in the [Serial Settings](#)^[224] topic). For the beep object, you just have to set the [beep.divider](#)^[388] correctly depending on the value returned by the sys.currpll property.

Data in the special configuration section of the EEPROM

Bottom 8 bytes of the EEPROM (accessible through the [stor](#)^[380] object) are reserved for storing a MAC address of the device. On power-up, the MAC address is read out from the EEPROM and programmed into the Ethernet controller. You can always check current MAC through the [net.mac](#)^[269] read-only property of the [net](#)^[267] object but there is no direct way to change it. Instead, you can change the MAC address data in the EEPROM. Then, next time the device boots up it will start using this new address.

By default, the area storing MAC address is not accessible to your application- the [stor.base](#)^[381] property takes care of that. Unless you change it, this property specifies that your application's storage area starts at address 9 (counting from 1). To change MAC, set the stor.base to 1.

MAC address data in the EEPROM has a certain formatting -- you have to follow it if you want the MAC to be recognized by the firmware (TiOS). Here is the format:

Byte1	Byte2	Byte3	Byte4	Byte5	Byte 6	Byte7	Byte8
6	MAC0	MAC1	MAC2	MAC3	MAC 4	MAC5	Checksum

Byte at address 1 must be set to 6- this means, that 6 byte of data follow (MAC address consists of 6 bytes). Addresses from 2 to 7 carry actual MAC data. Address 8 stores the checksum, which is calculated like this:

$255 - (\text{modulo8_sum_of_addr_1_through_7})$

Here is a sample code that stores new MAC address into the EEPROM and then reboots the device to make the device load this new MAC:

```

dim s as string
dim x as byte
...
s= "0.2.123.124.220.240" 'supposing, we want to set this MAC
...
...
s=chr(6)+ddval(s)      'added first byte (always 6) and converted
readable MAC into bytes
x=255-strsum(s)        'calculated checksum and assigned the
result to a BYTE variable (!!!)
s=s+chr(x)              'now our string is ready

stor.base(1)             'will access EEPROM from the bottom
x=stor.set(s,1)          'save data
if x<>len(s) then       'it is a good programming practice to check the
result
    'failed
else
    sys.reboot           'new MAC set, reboot!
end if

...

```

There are limitations on what MAC you can set. When loading the MAC into the Ethernet controller the device always resets the first byte of this address to 0. For example, if you set the MAC to 1.2.3.4.5.6 then the actual MAC used by the device will be 0.2.3.4.5.6.



If you write incorrect MAC data (wrong first byte or error in checksum calculation) the device will ignore it and boot up with default MAC, which is 0.1.2.3.4.100.

Available network interfaces

The DS1206 platform only has one network interface -- the Ethernet port. The [sock.allowedinterfaces](#)^[327] property refers to this interface as "NET".

Also see the [pl_sock_interfaces](#)^[145] enum which is used by [sock.targetinterface](#)^[359] and [sock.currentinterface](#)^[331] properties.

Miscellaneous

- [Sys](#)^[212] object supports the [sys.onsystemtimerperiod](#)^[220] property and the [on sys timer](#)^[220] event generation period depends on the value of this property.

Common Information

- [Supported Variable Types](#)^[183] (T1000-based devices)
- [Supported Functions](#)^[183] (T1000-based devices)
- [LED Signals](#)^[184]
- [Debug Communications](#)^[185]
- [Project Settings Dialog](#)^[186]

Supported Variable Types (T1000-based Devices)

The following variable types are supported by T1000-based devices:

- Byte
- Word
- Dword
- Char
- Short (integer)
- Long
- Real (float)
- Boolean
- User-defined structures
- User-defined enumeration types

For general type description see [Variables and Their Types](#)^[43].

Supported Functions (T1000-based Devices)

The following syscalls (platform functions) are supported by T1000-based devices:

- String-related:
 - [Asc](#)^[189] string character --> ASCII code;
 - [Chr](#)^[191] ASCII code --> string character;
 - [Val](#)^[210] numerical string--> 16-bit value (word or short);
 - [Lval](#)^[200] numerical string --> 32-bit value (dword or long);
 - [Strtof](#)^[209] numerical string --> real value;
 - [Bin](#)^[190] unsigned 16-bit numeric value (word) --> binary numerical string;
 - [Lbin](#)^[197] unsigned 32-bit numeric value (dword) --> binary numerical string;
 - [Str](#)^[207] unsigned 16-bit numeric value (word) --> decimal numerical string;
 - [Stri](#)^[208] signed 16-bit numeric value (short) --> decimal numerical string;
 - [Lstr](#)^[199] unsigned 32-bit numeric value (dword) --> decimal numerical string;
 - [Lstri](#)^[200] signed 32-bit numeric value (long) --> decimal numerical string;
 - [Hex](#)^[195] unsigned 16-bit numeric value (word) --> hexadecimal numerical string;
 - [Lhex](#)^[199]unsigned 32-bit numeric value (dword) --> hexadecimal numerical string;
 - [Ftostr](#)^[194] real value --> numerical string;
 - [Len](#)^[198] gets the string length;
 - [Left](#)^[198] gets a left portion of a string;
 - [Mid](#)^[202] gets a middle portion of a string;
 - [Right](#)^[205] gets a right portion of a string;
 - [Insert](#)^[196] inserts a string into another string;
 - [Instr](#)^[197]finds a substring in a string;
 - [Strgen](#)^[207] generates a string using repeating substring;
 - [Strsum](#)^[209] calculates 16-bit (word) sum of string characters' ASCII codes.

- [Ddstr](#)^[192] dot-decimal value --> dot-decimal string
- [Ddval](#)^[193] dot-decimal string --> dot-decimal value
- Date and time serialization and de-serialization:
 - [Daycount](#)^[192] given year, month, and date --> day number;
 - [Mincount](#)^[203] given hours and minutes --> minute number;
 - [Year](#)^[211] given day number --> year;
 - [Month](#)^[204] given day number --> month;
 - [Date](#)^[191] given day number --> date;
 - [Weekday](#)^[211] given day number --> day of the week;
 - [Hours](#)^[195] given minutes number --> hours;
 - [Minutes](#)^[203] given minutes number --> minutes.
- Hash calculation and related functions:
 - [Md5](#)^[201] calculates MD5 hash of a string;
 - [Sha1](#)^[205] calculates SHA-1 hash of a string;
 - [Random](#)^[205] generates a random string;
- Miscellaneous
 - [Cfloat](#)^[190] checks the validity of a real value.

LED Signals.3

BASIC-programmable devices supplied by Tibbo have two pairs of LEDs and/or control lines for external LEDs. The first pair is comprised of green and red status LEDs (SG and SR LEDs or lines), the second pair -- of green and yellow Ethernet status LEDs (EG and EY LEDs or lines).

Status LEDs have multiple functions:

- When the device is in the serial upgrade mode, these LEDs indicate the status of firmware upload process.
- When the device is under TiOS firmware control and Tibbo BASIC application is not running, these LEDs show current Tibbo BASIC application status.
- When the Tibbo BASIC application is running, status LEDs are under the control of [.pat](#)^[384] object (see <%T-B-P-G%>).

The following table summarizes predefined status LED patterns:

Serial upgrade mode		
	Green LED blinks slowly	File upload completed successfully.
	One long and one short "blink" of red LED	Communications error encountered during the serial file transfer.
	One long and two short "blinks" of red LED	FLASH memory failure.
Normal operation, Tibbo BASIC application not running		
	Fast-blinking GRGRGR... pattern	TiOS firmware not loaded or corrupted.

	Fast-blinking BBBB... pattern (B= red and green together)	Tibbo BASIC application loaded but cannot run due to insufficient variable (RAM) memory
	Fast-blinking G-G-G... pattern	Tibbo BASIC application loaded but not running.
	Fast-blinking R-R-R... pattern	Tibbo BASIC application not loaded or corrupted.

Ethernet status LEDs indicate the following:

- Link/Data LED (green) is turned on when "live" Ethernet cable is plugged into the device. The LED blinks whenever an Ethernet packet is received.
- 100BaseT LED (yellow) is turned on when the device links with the hub at 100Mb. The LED is off when the link is established at 10Mb.

Debug Communications

When designing the debug communications for exchanging debug information between the TIDE and the target, we had two important goals:

- Allow maximum flexibility for your application and occupy minimum resources on the target.
- Allow for fast and efficient debug communications.

The resulting debug communications system:

- Does not even require a "proper" IP address on the target side.
- Allows you to freely change the IP address of the target while debugging.
- "Occupies" a single UDP port (65535) on the target, and even this port can be used by your program in most cases.

Your PC running TIDE sends debug messages as broadcast UDP datagrams, to target's port 65535. These messages include the MAC address of the target on which you are debugging your application.

The UDP port 65535 can still be used by your BASIC application. The target recognizes a datagram received on this port as a debug command only if this datagram starts with an underscore (_).

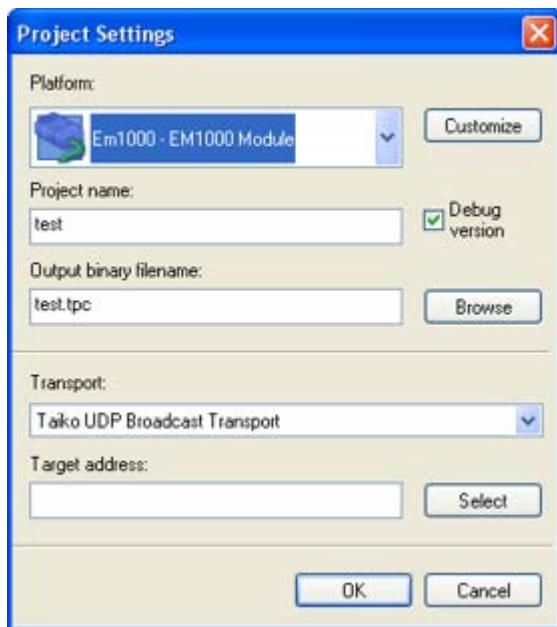
UDP datagrams received on UDP port 65535 that do not start with an underscore are not interpreted as debug commands by the target. Such datagrams are sent to your application for processing.



Broadcast cannot go across gateways (routers). This means the target and TIDE must reside on the same network segment -- remote debugging is not possible.

Project Settings Dialog

The [Project Settings](#)³⁸ dialog has been covered in the TIDE documentation.



Transport: leave "Taiko UDP Broadcast Transport" selected.

Target address: This is the MAC address of your target. As was explained in [Debug Communications](#)¹⁸⁵, the software uses the MAC address to address specific target device. Before you can upload and start debugging your application, you need to set correct MAC address of your target.

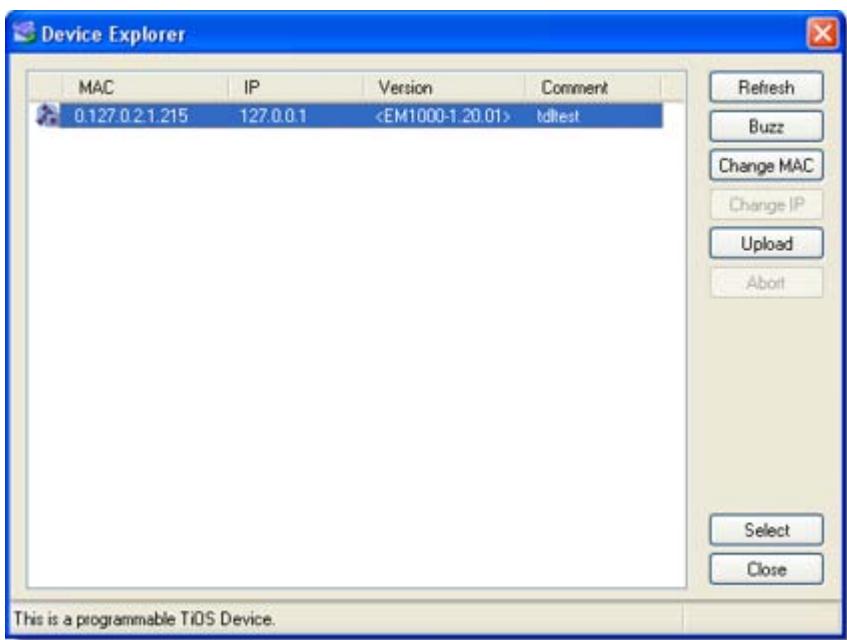
You can input the MAC address directly or click Browse to bring up the [Device Explorer](#)¹⁸⁷ dialog.

Customize Platform dialog

Pressing **Customize** button brings up the **Customize Platform** dialog. Your options depend on the platform and usually include enabling/disabling certain objects. For example, on the [EM1000](#)¹³⁹ platform you can enable/disable display support ([lcd](#)³⁹² object), flash disk support ([fd](#)⁴³³ object), and keypad support ([kp](#)⁴⁸⁴ object). Additionally, you get to choose the type of the display panel.

Disabling objects when they are not in use reduces the size of your compiled application binary.

Device Explorer



This dialog shows all targets found on the current network segment. A **target** is a hardware device capable of running TiOS.

Columns

Icon: The icon beside the MAC address shows the status of the target. It can be any of the following:

- The target is running fixed "Device Server" firmware, and has a valid IP address -- this status can never be displayed for the EM1000 device has never had any fixed firmware released for it.
- The target is running fixed "Device Server" firmware, and has an invalid (unreachable) IP address -- this status can never be displayed for the EM1000 device has never had any fixed firmware released for it.
- The target is running TiOS, and no BASIC application is in memory.
- The target is running TiOS, a BASIC application is in memory and is running.
- The target is running TiOS, a BASIC application is in memory and is stopped.
- The target is running TiOS, a BASIC application is in memory and is paused for debugging.

MAC: This is the current MAC address of the target. It can be changed.

IP: The current IP address of the target. It can be changed for targets running fixed (non-TiOS) firmware (does not apply to the EM1000). The IP address for a TiOS target should be changed via Tibbo BASIC code.

Version: The firmware version the device is running.

Comment: Additional information. For a device running fixed firmware, this is the owner name and the device name (does not apply to the EM1000). For a device running TiOS, this is the name of the Tibbo BASIC project currently loaded in memory.



You can click on a column header to sort the list by this column. This includes the status column (with the target icons). So you can sort the list according to devices which are currently running, waiting for a BASIC application, etc.

Buttons

Refresh: Check the network for any changes in targets, and update the list accordingly.

Buzz: Make the LEDs of the currently selected targets flash, or stop flashing (depending on current firmware). This is useful for physically locating a target when several are in sight.

Change MAC: Change the MAC address of the target. Note that you have to **reboot** the target (power it off and back on) for the change to become effective.

Change IP: Change the IP address of the target. This option is only available when the target is running fixed firmware (not TiOS).

Upload: This is a 'flyout' button. Clicking it opens a menu with the following commands:

Load Firmware Through the Network: Upload firmware (fixed or TiOS) onto target(s).

Load Application Through the Network: Upload a Tibbo BASIC application onto target(s) running TiOS.

Load Firmware Through the Serial Port: Upload firmware (fixed to TiOS) onto a single target via the serial port of the computer (COM1, COM2, etc).

Abort: Aborts the current operation. Operations in this dialog do not time out -- you must abort them manually. This is beneficial, since if there is any problem you may troubleshoot it, and once resolved, operation will just resume normally -- you will not have to restart a complex process or restore some state.

Select: This button is only visible when accessing the dialog from within TIDE. Once a specific target has been selected, the currently open project in TIDE will use it for cross-debugging.

Close: Close the dialog.

Highlighting Multiple Targets

You may highlight multiple targets in the list by left-clicking on a blank space in the list and dragging ("lasso"), or by holding SHIFT while pressing the arrow keys to move down or up the list.

This allows you to perform certain operations en masse -- such as uploading TiOS to an entire group of devices, or deploying a Tibbo BASIC program on multiple devices, etc.

Stand-Alone Mode

This dialog was designed so it could be accessed also without running TIDE, or indeed, without TIDE even being installed on the system. This is useful in multi-tiered environments. If a vendor writes an application, he can then provide the customer with a .bin (firmware) or .tpc (compiled PCode - BASIC application) file or to load onto the targets and with a stand-alone version of the dialog. The customer does not have to have TIDE installed, and can easily administer the

targets on his network.

To access the dialog in stand-alone mode, run em202upgr.vbs.

If You Cannot See Your Target

There can be several possible causes why you cannot see your target. Two of the most common ones are:

1) The target is not connected to the same subnet as the computer. I.e., there is a router between the computer and the DS. **To fix this:** Connect the DS to the same hub as the computer.

2) There is a local firewall installed on the computer. Local firewalls usually block broadcast UDP datagrams, which are used to communicate with the target.

To fix this: Configure the firewall to allow TIDE to send broadcast UDP datagrams.



Of course, once you upload a device with the TiOS firmware, it is no longer a Device Server! So you cannot see it under DS Manager. You could program it so it would respond to DS Manager -- but by default it is a 'clean slate', and does not respond to DS Manager broadcasts.

Function Reference

Function reference section is a repository for all functions (syscalls) that have ever been created. The platform you are working with does not necessarily support every function. See your [platform specifications](#)^[133] – you will find the list of support functions there.

Asc Function

Function: Returns the ASCII code of the leftmost character of the string.

Syntax: **asc(byref sourcestr as string) as byte**

See Also: [Chr](#)^[191]

Part	Description
sourcestr	Input string; the function will return ASCII code of the leftmost character of this string.

Details

Examples

```
x = asc("123") ' result will be 49 (ASCII code of '1')
```

Bin Function

Function:	Converts unsigned 16-bit numeric value (word) into its binary string representation.
Syntax:	bin(num as integer) as string
See Also:	str ²⁰⁷ , lstr ¹⁹⁹ , stri ²⁰⁸ , lstri ²⁰⁰ , lbin ¹⁹⁷ , hex ¹⁹⁵ , lhex ¹⁹⁹ , val ²¹⁰ , lval ²⁰⁰

Part	Description
num	Value to convert.

Details

Standard "&b" prefix is added at the beginning of the string.

Examples

```
dim s as string
s = bin(34) ' result will be '&b100010'
```

Cfloat Function

Function:	Vерифицирует значение переменной с плавающей точкой. Возвращает 0 - VALID, если значение является корректным, и 1 - INVALID, если значение является некорректным.
Syntax:	cfloat(byref num as real) as valid_invalid
See Also:	strtof ²⁰⁹ , ftosrt ¹⁹⁴

Part	Description
num	Variable to check.

Details

Floating-point calculations can lead to invalid result (#INF, -#INF errors, as per IEEE specification). When your application is in the [debug](#)²⁷ mode you will get a [FPERR exception](#)²⁹ if such an error is encountered. In the release mode the Virtual Machine won't generate an exception, yet your application may need to know if a certain floating-point variable contains correct value. This is where cfloat function comes handy.

Examples

```

dim r1 as real
dim v as invalid_valid

dim r1=10E30
v=cffloat(r1) 'v will return 0- VALID
r1=r1*10E20 'at this point you will get FPERR exception if you are in the
debug mode
v=cffloat(r1) 'v will return 1- INVALID

```

Chr Function

Function:	Returns the string that consists of a single character with ASCII code asciicode.
Syntax:	chr(asciicode as byte) as string
See Also:	Asc ^[189]

Part	Description
asciicode	ASCII code of the character to return.

Details

It is also possible to use this function within string definitions, as shown in the example below.

Examples

```

dim x as byte
dim s as string

x=49
s = chr(x) ' result will be '1'
s = "FooBar" + chr(13) ' would add a carriage return to the end of the
string

```

Date Function

Function:	Returns the date for a given day number.
Syntax:	date(daycount as word) as byte
Returns:	Date in the 1-31 range.
See Also:	year ^[211] , month ^[204] , weekday ^[211] , daycount ^[192] , hours ^[195] , minutes ^[203] , mincount ^[203]

Part	Description
------	-------------

daycount Day number. Base date for the day count is 1-JAN-2000 (this is day #0).

Details

Examples

```
b = day(366) ' result will be 1 -- because day 366 is actually January
1st, 2001.
```

Daycount Function

Function:	Returns the day number for a given year, month, and date.
Syntax:	daycount(year as byte, month as byte, date as byte) as word
Returns:	Day number elapsed since 1-JAN-2000 (this is day #0). The range is 0-36524.
See Also:	year ^[211] , month ^[204] , date ^[191] , weekday ^[211] , hours ^[195] , minutes ^[203] , mincount ^[203]

Part	Description
year	The year is supplied as offset from year 2000 (so, it is 6 for year 2006). Acceptable year range is 0-99 (2000-2099).
month	1-12 for January-December
date	Date of the month

Details

If any input parameter is illegal (year exceeds 99, month exceeds 12, etc.) this syscall will return 65535. This error value cannot be confused with an actual valid day number since the maximum day number recognized by this syscall is 12-DEC-2099 (day number 36524).

Examples

```
w = daycount(06, 10, 15) ' result will be 2479 (the serial day number for
October 15th, 2006)
```

Ddstr Function

Function:	Converts "dot-decimal value" into "dot-decimal string".
Syntax:	ddstr(byref str as string) as string

Returns: A dot-separated string consisting of decimal representations of all binary values in the input string. Each decimal value will be in the 0-255 range.

See Also: [ddval](#)

Part	Description
str	String of binary values to be converted into a dot-decimal string.

Details

This function is convenient for converting groups of bytes representing binary data (such as IP or MAC addresses) into their string representation.

Examples

```
dim s as string
s = chr(192)+chr(168)+chr(100)+chr(40) 'produce a string that contains
these values: 192,168,100,40
s = ddstr(s) 'now s will be equal to "192.168.100.40"
```

Ddval Function

Function: Converts "dot-decimal string" into "dot-decimal value".

Syntax: **ddval (byref str as string) as string**

Returns: A string of binary values.

See Also: [ddstr](#)

Part	Description
str	Dot-decimal string to be converted into a string of binary values. This string should comprise one or more dot-separated decimal values in the 0-255 range. Values that exceed 255 will produce an overflow, so result will be incorrect. If any other character other than "0"- "9" or "." is encountered then all digits after this character and up to the next "." (if any) will be ignored. Leading spaces before each decimal value are allowed.

Details

This function is convenient for converting string representation of groups of bytes (such as IP or MAC addresses) into their binary form.

Examples

```

dim s as string

s = "192_3.1254.. 30" 'One value has invalid character in it ("_") and "3"
after this character will be ignored. Dot-decimal string. Another value --
1254 -- is out of range. Yet another value is missing and will be replaced
with 0.
s = ddstr(s) 'now s will contain these values: 192, 230, 0, 30.

```

Ftostr Function

Function:	Converts real value into its string representation.
Syntax:	ftostr(byref num as real, mode as ftostr_mode, rnd as byte) as string
See Also:	strtof ^[209] , cfloat ^[190] , str ^[207] , val ^[210]

Part	Description
num	Real value to convert.
mode	Desired output format: 0- FTOSTR_MODE_AUTO: Choose between plain and mantissa/exponent format automatically. If mantissa/exponent format results in shorter string it will be used, otherwise plain format will be used. 1- FTOSTR_MODE_ME: Use mantissa/exponent format. 2- FTOSTR_MODE_PLAIN: Use plain format, not mantissa/exponent representation.
rnd	Number of digits to round the result to (total number of non-zero digits in the integer and fractional part of mantissa).

Details

Ftostr function offers much more control over the format of the output string compared to similar functions found on other systems. For starters, you can select whether you want to see mantissa/exponent, "regular" format, or let the function decide which format to use. Additionally, you control the rounding i.e. get to choose how many digits should be displayed -- and this influences the representation both of the fractional and integer part of the value.

Examples below illustrate what you can do with ftostr. The ftostr has a counterpart -- fstr -- which is invoked implicitly whenever you assign a real to a string (string=real). Fstr is just like ftostr but mode and rnd parameters are fixed at 0-FTOSTR_MODE_AUTO and "maximum number of digits possible".

Examples

```

dim r1 as real
dim s as string

'demonstrate output formats
r1=10000000000.0 'notice '.0' -- it is necessary or compilier will
generate an error
s=ftostr(r1,FTOSTR_MODE_ME,11) 'result will be '1E+010'
s=ftostr(r1,FTOSTR_MODE_PLAIN,11) 'result will be '10000000000'
s=ftostr(r1,FTOSTR_MODE_AUTO,11) 'result will be '1E+010' because this
representation is more
compact

'demonstrate rounding
r1=1234567.125
s=ftostr(r1,FTOSTR_MODE_AUTO,15) 'result will be '1234567.125'
s=ftostr(r1,FTOSTR_MODE_AUTO,9) 'result will be '1234567.13'
s=ftostr(r1,FTOSTR_MODE_AUTO,2) 'result will be '1200000'

s=r1 'fstr will be used, result will be '1234567.125'

```

Hex Function

Function: Converts unsigned 16-bit numeric value (word) into its HEX string representation.

Syntax: **hex(num as integer) as string**

See Also: [str](#)^[207], [lstr](#)^[199], [stri](#)^[208], [lstri](#)^[200], [bin](#)^[190], [lbin](#)^[197], [lhex](#)^[199], [val](#)^[210], [lval](#)^[200]

Part	Description
num	Value to convert.

Details

Standard "&h" prefix is added at the beginning of the string.

Examples

```

dim s as string
s = hex(34) 'result will be '&h22'

```

Hours Function

Function: Returns the hours value for a given minutes number.

Syntax: **hours (mincount as word) as byte**

Returns: Hours in the 0-23 range.

See Also: [year](#)^[211], [month](#)^[204], [date](#)^[191], [weekday](#)^[211], [daycount](#)^[192], [minutes](#)^[203], [mincount](#)^[203]

Part	Description
mincount	The number of minutes elapsed since midnight (00:00 is minute #0). Maximum mincount number is 1439 (23:59).

Details

If a value higher than 1439 is supplied, this call will return 255. This error value cannot be confused with valid output since normal hours value cannot exceed 23.

Examples

```
b = hours(620) ' result will be 10 (because this minute number is falls between 10:00 and 11:00)
```

.Insert Function

Function: Inserts insert_str string into the dest_str string at the insert position pos. Returns the new length of dest_str.

Syntax: **insert(byref dest_str as string, pos as byte,byref insert_str as string) as byte**

See Also: ---

Part	Description
insert_str	The string to insert.
pos	Insert position in the dest_str.
dest_str	The string to insert into.

Details

This is an insert with overwrite, meaning that the insert_str will overwrite a portion of the dest_str.

Dest_str length can increase as a result of this operation (but not beyond declared string capacity). This will happen if the insertion position does not allow the source_str to fit within the current length of the dest_string.

Examples

```
s = "FLIGHT XXX STATUS"
insert("123",8,s) 's will now be 'FLIGHT 123 STATUS'
```

Instr Function

Function: Finds the Nth occurrence (defined by num, counting from 1) of a substring substr in a string sourcestr. Search is conducted from position frompos (leftmost character has position 1).

Syntax: **instr(frompos as byte,byref sourcestr as string, byref substr as string, num as byte) as byte**

See Also: ---

Part	Description
frompos	Position in the sourcestr from which to start searching. Leftmost character has position 1.
sourcestr	Source string in which the substring is to be found.
substr	Substring to search for within the source string.
num	Occurrence number of the substr, counting from 1.

Details

This function returns position in a string or zero if the Nth occurrence of the substring is not found.

Examples

```
x = substr(3, "ABCABCDEABC12", "BC", 2) ' result will be 10
```

Lbin Function

Function: Converts unsigned 32-bit numeric value (dword) into its binary string representation.

Syntax: **lbin(byref num as dword) as string**

See Also: [str](#)^[207], [lstr](#)^[199], [stri](#)^[208], [lstri](#)^[200], [bin](#)^[190], [hex](#)^[195], [lhex](#)^[199], [val](#)^[210], [lval](#)^[200]

Part	Description
num	Value to convert.

Details

Standard "&b" prefix is added at the beginning of the string.

Examples

Left Function

Function:	Returns len leftmost characters of a string sourcestring.
Syntax:	left(byref sourcestr as string, len as byte) as string
See Also:	Right ²⁰⁵ , Mid ²⁰²

Part	Description
sourcestr	String from which to take <i>len</i> leftmost characters.
len	Number of characters to take.

Details

- - -

Examples

```
ss = left("ABCDE", 3)  ' result will be 'ABC'
```

Len Function

Function:	Returns the length of (number of characters in) the string sourcestr.
Syntax:	len(byref sourcestr as string) as byte
See Also:	---

Part	Description
sourcestr	String whose length needs to be calculated.

Details

- - -

Examples

```
x = len("ABC")  ' result will be 3
```

Lhex Function

Function: Converts unsigned 32-bit numeric value (dword) into its HEX string representation.

Syntax: **Ihex(byref num as dword) as string**

See Also: [str](#)^[207], [Istr](#)^[199], [stri](#)^[208], [Istri](#)^[200], [bin](#)^[190], [Ibin](#)^[197], [hex](#)^[195], [val](#)^[210], [Ival](#)^[200]

Part	Description
------	-------------

num Value to convert.

Details

Standard "&h" prefix is added at the beginning of the string.

Examples

```
dim s as string
s = hex(65536) 'result will be '&h10000'
```

Lstr Function

Function: Converts unsigned 32-bit numeric value (dword) into its decimal string representation.

Syntax: **Istr(byref num as dword) as string**

See Also: [str](#)^[207], [stri](#)^[208], [Istri](#)^[200], [bin](#)^[190], [Ibin](#)^[197], [hex](#)^[195], [Ihex](#)^[199], [val](#)^[210], [Ival](#)^[200]

Part	Description
------	-------------

num Value to be converted to string.

Details

Can be invoked implicitly, through the *string_var= dword_var* expression (see example below). Compiler is smart enough to pre-calculate constant-only expressions involving implicit use of Istr function.

Examples

```

dim d as dword
dim s as string

d1=123456
s = lstr(d) 'explicit invocation. Result will be '123456'
s = d 'implicit invocation
s = 666666 'will be calculated at compilation -- no actual lstr function
invocation will be here

```

Lstri Function

Function: Converts signed 32-bit numeric value (long) into its decimal string representation.

Syntax: **Istri(byref num as long) as string**

See Also: [str](#)^[207], [Istr](#)^[199], [stri](#)^[208], [bin](#)^[190], [Ibin](#)^[197], [hex](#)^[195], [Ihex](#)^[199], [val](#)^[210], [Ival](#)^[200]

Part	Description
num	Value to be converted to string.

Details

Can be invoked implicitly, through the *string_var=long_var* expression (see example below). Compiler is smart enough to pre-calculate constant-only expressions involving implicit use of Istri function.

Examples

```

dim l as long
dim s as string

l= -5123
s = lstri(l) 'explicit invocation.
s = l 'implicit invocation
s = lstri(-20) 'will be calculated at compilation -- no actual lstri
function invocation will be here

```

Lval Function

Function: Converts string representation of a value into 32-bit value (dword or long).

Syntax: **Ival(byref sourcestr as string) as dword**

See Also: [str](#)^[207], [Istr](#)^[199], [stri](#)^[208], [Istri](#)^[200], [bin](#)^[190], [Ibin](#)^[197], [hex](#)^[195], [Ihex](#)^[199], [val](#)^[210]

Part	Description
sourcestr	String to convert.

Details

Recognizes &b (binary) and &h (hexadecimal) prefixes. Can be invoked implicitly, through the `dword_var= string_var` expression (see example below). Compiler is smart enough to pre-calculate constant-only expressions involving implicit use of `lval` function.

Examples

```
dim d as word
dim l as long
dim s as string

s = "4294967295"
d = lval(s) 'explicit invocation, result will be 4294967295
l = lval(s)
d = s 'implicit invocation, result will be -1 (4294967295 -> &hFFFFFFFF ->
-1 for signed variable)
d = "2402" 'be calculated at compilation -- no actual lval function
invocation will be here
```

Md5 Function

Function:	Generates MD5 hash on the str string.
Syntax:	md5(byref str as string,byref input_hash as string, md5_mode as md5_modes,total_len as word) as string
Returns:	16-character hash string; an empty string when invalid str or input_hash argument was detected
See Also:	sha1

Part	Description
str	String containing (the next portion of) the input data to generate MD5 hash on. When <code>md5_mode= 0-MD5_UPDATE</code> , this string must be 64, 128, or 192 characters in length. Any other length will result in error and the function will return an empty string. When <code>md5_mode= 1- MD5_FINISH</code> , this string can have any length (up to 255 bytes).
input_hash	Hash obtained as a result of MD5 calculation on the previous data portion. Leave it empty for the first portion of data. Use the result of MD5 calculation on the previous data portion for the second and all subsequent portions of data. The result of MD5 is always 16 characters long, so passing the string of any other length (except 0 -- see above) will result in error and this function will return an empty string.

md5_mode	0- MD5_UPDATE : Set this mode for all data portions except the last one. 1- MD5_FINISH: Set this mode for the last data portion; also use this selection if you only have a single data portion.
total_len	Total length of processed data (in all data portions combined). Only relevant when md5_mode= 1-MD5_FINISH. That is, only relevant for the last or a single data portion.

Details

MD5 is a standard method of calculating hash codes on data of any size. The amount of input data can often exceed maximum capacity of string variables (255 characters). The md5 method can be invoked repeatedly in order to process the data of any size (see the example below).

Examples

```
Dim s, hash As String

'simple calculation on a short string
s="String to calculate MD5 on"
hash=md5(s,"",MD5_FINISH,Len(s))

'calculation on the entire contents of data in the 'text.txt 'file
romfile.open("text.txt")
hash=""
s=romfile.getdata(192) 'use max portions
While Len(s)=192
    hash=md5(s,hash,MD5_UPDATE,0)
    s=romfile.getdata(192)
Wend
hash=md5(s,hash,MD5_FINISH,romfile.size) 'last portion for whatever
unprocessed data is remaining in the file
```

Mid Function

Function: Returns len characters from a string sourcestr starting from position pos.

Syntax: **mid(byref sourcestr as string, frompos as byte, len as byte) as string**

See Also: [Left](#)^[198], [Right](#)^[205]

Part	Description
sourcestr	String from which to take the middle section.
frompos	First character to take. The leftmost character is counted to be at position 1.
len	Number of characters to take.

Details

Examples

```
s = mid("ABCDE", 2, 3) ' result will be 'BCD'.
```

Mincount Function

Function:	Returns the minutes number for a given hours and minutes.
Syntax:	mincount(hours as byte, minutes as byte) as word
Returns:	Minute number elapsed since midnight (00:00). This value is in the 0-1439 range.
See Also:	year ^[211] , month ^[204] , date ^[191] , weekday ^[211] , daycount ^[192] , hours ^[195] , minutes ^[203]

Part	Description
hours	An hour value, from 0 to 23.
hours	A minute value, from 0 to 59.

Details

If any input parameter is illegal (hours exceeds 23, minutes exceeds 59, etc.) this syscall will return 65535. This error value cannot be confused with an actual valid minute number since the maximum minute number cannot exceed 1439.

Examples

```
w = mincount(14, 00) ' result will be 840
```

Minutes Function

Function:	Returns the minutes value for given minutes number.
Syntax:	minutes(mincount as word) as byte
Returns:	Minutes in the 0-59 range.
See Also:	year ^[211] , month ^[204] , date ^[191] , weekday ^[211] , daycount ^[192] , hours ^[195] , mincount ^[203]

Part	Description
mincount	The number of minutes elapsed since midnight (00:00 is minute #0).

Details

If a value higher than 1439 is supplied, this call will return 255. This error value cannot be confused with valid output since normal minutes value cannot exceed 59.

Examples

```
b = minutes(61) ' result will be 1 - this is the time 01:01.
```

Month Function

Function:	Returns the month for a given day number.
Syntax:	month(daycount as word) as pl_months
Returns:	One of pl_months constants: 1- PL_MONTH_JANUARY: January. 2- PL_MONTH_FEBRUARY: February. 3- PL_MONTH_MARCH: March. 4- PL_MONTH_APRL: April. 5- PL_MONTH_MAY: May. 6- PL_MONTH_JUNE: June. 7- PL_MONTH_JULY: July. 8- PL_MONTH_AUGUST: August. 9- PL_MONTH_SEPTEMBER: September. 10- PL_MONTH_OCTOBER: October. 11- PL_MONTH_NOVEMBER: November. 12- PL_MONTH_DECEMBER: December.

See Also: [year](#)^[21], [date](#)^[19], [weekday](#)^[21], [daycount](#)^[192], [hours](#)^[195], [minutes](#)^[203], [mincount](#)^[203]

Part	Description
daycount	Day number. Base date for the day count is 1-JAN-2000 (this is day #0).

Details

Examples

```
dim m as pl_months
m = month(32)  ' result will be PL_MONTH_FEBRUARY - day number 32 was in
February 2000.
```

Random Function

Function: Generates a string consisting of len random characters.

Syntax: **random(len as byte) as string**

See Also: ---

Part	Description
len	Length of the string to generate.

Details

Right Function

Function: Returns len rightmost characters of a string sourcestr.

Syntax: **right(byref sourcestr as string, len as byte) as string**

See Also: [Left](#)¹⁹⁸, [Mid](#)²⁰²

Part	Description
sourcestr	String from which to take <i>len</i> rightmost characters.
len	Number of characters to take.

Details

Examples

```
s = right("ABCDE",3)  ' result will be 'CDE'
```

Sha1 Function

Function: Generates SHA1 hash on the str string.

Syntax: **sha1(byref str as string,byref input_hash as string, sha1_mode as sha1_modes, total_len as word) as string**

Returns: 20-character hash string; an empty string when invalid str or input_hash argument was detected

See Also: [md5](#)

Part	Description
str	String containing (the next portion of) the input data to generate SHA1 hash on. When sha1_mode= 0-SHA1_UPDATE, this string must be 64, 128, or 192 characters in length. Any other length will result in error and the function will return an empty string. When sha1_mode= 1- SHA1_FINISH, this string can have any length (up to 255 bytes).
input_hash	Hash obtained as a result of SHA1 calculation on the previous data portion. Leave it empty for the first portion of data. Use the result of SHA1 calculation on the previous data portion for the second and all subsequent portions of data. The result of sha1 is always 20 characters long, so passing the string of any other length (except 0 -- see above) will result in error and this function will return an empty string.
sha1_mode	0- SHA1_UPDATE : Set this mode for all data portions except the last one. 1- SHA1_FINISH: Set this mode for the last data portion; also use this selection if you only have a single data portion.
total_len	Total length of processed data (in all data portions combined). Only relevant when sha1_mode= 1-SHA1_FINISH. That is, only relevant for the last or a single data portion.

Details

SHA1 is a standard method of calculating hash codes on data of any size. The amount of input data can often exceed maximum capacity of string variables (255 characters). The sha1 method can be invoked repeatedly in order to process the data of any size (see the example below).

Examples

```

Dim s, hash As String

'simple calculation on a short string
s="String to calculate SHA1 on"
hash=sha1(s,"",SHA1_FINISH,Len(s))

'calculation on the entire contents of data in the 'text.txt 'file
romfile.open("text.txt")
hash=""
s=romfile.getdata(192) 'use max portions
While Len(s)=192
    hash=sha1(s,hash,SHA1_UPDATE,0)
    s=romfile.getdata(192)
Wend
hash=sha1(s,hash,SHA1_FINISH,romfile.size) 'last portion for whatever
unprocessed data is remaining in the file

```

Str Function

Function: Converts unsigned 16-bit numeric value (word) into its decimal string representation.

Syntax: **str(num as word) as string**

See Also: [lstr](#)^[199], [stri](#)^[208], [lstri](#)^[200], [bin](#)^[190], [lbin](#)^[197], [hex](#)^[195], [lhex](#)^[199], [val](#)^[210], [lval](#)^[200]

Part	Description
num	Value to be converted to string.

Details

Can be invoked implicitly, through the *string_var= word_var* expression (see example below). Compiler is smart enough to pre-calculate constant-only expressions involving implicit use of str function.

Examples

```

dim w as word
dim s as string

w=3400
s = str(w) 'explicit invocation.
s = w 'implicit invocation
s = 100 'will be calculated at compilation -- no actual str function
invocation will be here

```

Strgen Function

Function: Generates a string of len length consisting of repeating substrings substr.

Syntax: **strgen(len as byte,byref substr as string) as string**

See Also: ---

Part	Description
len	Length of the string to generate
substr	Substring that will be used (repeatedly) to generate the string

Details

Notice that len parameter specifies total resulting string length in bytes so the last substring will be truncated if necessary to achieve exact required length. This function is an expanded version of the STRING\$ function commonly found in other BASICs.

Examples

```
string1 = strgen(10, "ABC") ' result will be 'ABCABCABCA'.
```

Stri Function

Function: Converts signed 16-bit numeric value (short) into its decimal string representation.

Syntax: **stri(num as integer) as string**

See Also: [str](#)^[207], [lstr](#)^[199], [lstri](#)^[200], [bin](#)^[190], [lbin](#)^[197], [hex](#)^[195], [lhex](#)^[198], [val](#)^[210], [lval](#)^[200]

Part	Description
num	Value to be converted to string.

Details

Can be invoked implicitly, through the *string_var= short_var* expression (see example below). Compiler is smart enough to pre-calculate constant-only expressions involving implicit use of stri function.

Examples

```
dim sh as short
dim s as string

sh= -3400
s = stri(sh) 'explicit invocation.
s = sh 'implicit invocation
s = stri(-100) 'will be calculated at compilation -- no actual stri
function invocation will be here
```

Strsum Function

Function: Calculates 16-bit (word) sum of ASCII codes of all characters in a string.

Syntax: **strsum(byref sourcestr as string) as word**

See Also: ---

Part	Description
sourcestr	String to work on

Details

This function is useful for checksum calculation.

Examples

```
w = strsum("012")    ' will return 147 (48+49+50).
```

Strtof Function

Function: Converts string representation of a real value into a real value.

Syntax: **strtof(byref str as string) as real**

See Also: [ftostr](#)^[194], [str](#)^[207], [val](#)^[210]

Part	Description
str	String to convert.

Details

You must keep in mind that floating-point calculations are inherently imprecise. Not every value can be converted into its exact floating-point representation. Also, strtof can be invoked implicitly. Examples below illustrates this.

Examples

```

dim r1 as real
dim s as string

s="456.125"
r1=strcmp(s) 'r1 will be equal to 456.125. This conversion will be done
without errors.
s="123.200"
r1=strcmp(s) '123.200 will be converted with errors. Actual result will be
123.25.
r1=s 'implicit invocation. Same result as for the line above.

```

Val Function

Function: Converts string representation of a value into 16-bit value (word or short).

Syntax: **val (byref sourcestr as string) as word**

See Also: [str](#)^[207], [Istr](#)^[199], [stri](#)^[208], [Istri](#)^[200], [bin](#)^[190], [Ibin](#)^[197], [hex](#)^[195], [Ihex](#)^[199], [Ival](#)^[200]

Part	Description
sourcestr	String to convert.

Details

Recognizes &b (binary) and &h (hexadecimal) prefixes. Can be invoked implicitly, through the *word_var = string_var* expression (see example below). Compiler is smart enough to pre-calculate constant-only expressions involving implicit use of val function.

Beginning with release **2.0**, this function also plays the role of vali function, which has been removed.

Examples

```

dim w as word
dim sh as short
dim s as string

s="&hF222"
w = val(s) 'explicit invocation, result will be 61986
sh = val(s) 'explicit invocation, result will be -3550 (sh is a 16-bit
signed variable)
w = s 'implicit invocation
w = "2402" 'be calculated at compilation -- no actual val function
invocation will be here

```

Vali Function

Vali function is no longer available. Use [val](#)^[210] function both for unsigned (word) and signed (short) conversions.

Weekday Function

Function:	Returns the day of the week for a given day number.
Syntax:	weekday(daycount as word) as pl_days_of_week
Returns:	One of pl_days_of_week constants: 1- PL_DOW_MONDAY: Monday. 2- PL_DOW_TUESDAY: Tuesday. 3- PL_DOW_WEDNESDAY: Wednesday. 4- PL_DOW_THURSDAY: Thursday. 5- PL_DOW_FRIDAY: Friday. 6- PL_DOW_SATURDAY: Saturday. 7- PL_DOW_SUNDAY: Sunday.

See Also: [year](#)^[211], [month](#)^[204], [date](#)^[191], [daycount](#)^[192], [hours](#)^[195], [minutes](#)^[203], [mincount](#)^[203]

Part	Description
daycount	Day number. Base date for the day count is 1-JAN-2000 (this is day #0).

Details

Examples

```
dim w as pl_days_of_week
w = weekday(0)    result will be  PL_DOW_SATURDAY - the was the day of the
                                week for the 1st of January 2000.
```

Year Function

Function:	Returns the year for a given day number.
Syntax:	year(daycount as word) as byte
Returns:	Two last digits of the year (0 means 2000, 1 means 2001, and so on.)
See Also:	month ^[204] , date ^[191] , weekday ^[211] , daycount ^[192] , hours ^[195] , minutes ^[203] , mincount ^[203]

Part	Description
------	-------------

daycount	Day number. Base date for the day count is 1-JAN-2000 (this is day #0).
----------	---

Details

Examples

```
b = year(366) ' result will be 1 (this day number is in year 2001).
```

Object Reference

Object reference section is a repository for all objects that have ever been created. The platform you are working with does not necessarily support every object. See your [platform specifications](#)^[133]- you will find the list of supported objects there.

Sys Object



This is the system object that loosely combines "general system" stuff such as initialization (boot) event, buffer management, system timer, PLL mode, and some other miscellaneous properties and methods.

Overview

Here you will find:

- [On_sys_init \(boot\) event](#)^[212]
- [Buffer Management](#)^[213]
- [System Timer](#)^[214]
- [PLL Management](#)^[215]
- [Serial Number](#)^[216]
- [Miscellaneous stuff](#)^[217] such as how to halt execution, reboot, check firmware version.

On_sys_init Event

The sys object provides a very important event- [on_sys_init](#)^[220]. This event is guaranteed to be generated first when your device starts running. Therefore, you should put all your initialization code for sockets, ports, .etc into the event handler for this event:

```

sub on_sys_init
    net.ip="192.168.1.95"
    sock.num=0
    sock.targetip= "192.168.1.96"
    '...and everything else that you may need!
end sub

```

Buffer Management

How to allocate buffer memory

Some objects, such as the [sock](#)^[274] or [ser](#)^[224], rely on buffers for storing the data being sent and received. For example, each serial port of the ser object has two buffers- RX and TX, while each socket of the sock object has 6 different buffers.

By default, all buffers have no memory allocated for them, which basically means that they cannot store any data. For related objects to work, these buffers need to be given memory.

Memory is allocated in pages. Each page equals 256 bytes. There is a small overhead for each buffer- 16 bytes are used for internal buffer housekeeping (variables, pointers, etc.). Therefore, if you have allocated 2 pages for a particular buffer, then this buffer's actual capacity will be $2 \times 256 - 16 = 496$ bytes.

Buffer memory allocation is a two-step process. First, you request certain number of pages for each port, socket, etc. that you plan to use. This is done through methods of corresponding objects. For example, the [ser.rxbuffrq](#)^[262] method requests buffer memory for the RX buffer of one of the serial ports. Similar methods exist for all other buffers of all objects that require buffer memory. Note, that actual memory allocation does not happen at this stage- only your "requests" are collected.

After all requests have been made actual memory allocation is performed by using the [sys.buffalloc](#)^[217] method. This allocates memory, as per previous requests, for all buffers of all objects. Here is an example:

```

'allocate memory for RX and TX buffers of serial port 0 and socket 0

'make requests
ser.num= 0
ser.rxbuffrq= 5
ser.txbuffrq= 5
sock.num=0
sock.rxbuffrq= 4
sock.txbuffrq= 4

'and now actual allocation
sys.buffalloc

```

Typically, buffer memory allocation is done in the on_sys_init event handler but you don't have to do it this way. In fact, your application can re-allocate buffer memory space according to the changes in the operating mode or other conditions.

Sys.buffalloc could take up to several hundred milliseconds to execute, so it makes sense to use it as little as possible. Hence, request all necessary buffer allocations first, then use the sys.buffalloc once to finish the job.

Buffer (re)allocation for a specific object will only work if the corresponding object or part of the object to which this buffer belongs is idle. "Part" refers to a particular

serial port of the ser object, or particular socket of the sock object, etc. to which the buffer you are trying to change belongs. "Idle" means different things for different objects: [ser.enabled](#)^[25]= 0- NO for the serial port, [sock.statesimple](#)^[358]= 0- PL_SSTS_CLOSED for the socket, etc.

Intelligent memory allocation basing on what's available

Memory is not an infinite resource and you will not always get as much memory as you have requested. Methods such as the [ser.rxbuffrq](#)^[262] actually return the amount of memory that can be allocated:

```
dim x as byte
x=ser.rxbuffrq(5) 'x could get less than 5, which means that you got less
memory than you have asked for
```

Two properties of the sock object were implemented to help you allocate memory intelligently, basing on what is available.

The [sys.totalbuffpages](#)^[223] read-only property tells you how many buffer pages are there on your system in total. This is defined by the amount of physical variable memory (RAM) available minus memory required to store your project's variables (so, as your project grows available buffer memory shrinks).

The [sys.freebuffpages](#)^[218] read-only property tells you the amount of free (not yet allocated) buffer pages.

Here is an example in which we allocate memory equally between the TX and RX buffers of four serial ports and four sockets:

```
dim x,f as byte

x=sys.totalbuffpages/16 'will work correctly if we haven't allocated
anything yet

for f=0 to 3
    ser.num= f
    ser.rxbuffrq(x)
    ser.txbuffrq(x)
    sock.num= f
    sock.rxbuffrq(x)
    sock.txbuffrq(x)
next f
```

In the above example we do not check what individual buffer requests return because we have already calculated each buffer's size basing on the total number of buffer pages available.

System Timer

The system object provides a timer event- [on sys timer](#)^[220]- that is generated periodically. The period depends on the platform. If the platform doesn't have a [sys.onsystimerperiod](#)^[220] property, then this period is fixed at 0.5 seconds. If the

platform supports this property, then 0.5 second is the default period and the property allows your program to adjust it.

If your software has some periodic tasks you can put the code into the `on_sys_timer` event handler. Notice that when generated, this event goes into the queue and waits in line to be processed, just like all other events. Therefore, you cannot expect great accuracy in the period at which the `on_sys_timer` event handler is entered. You can just expect that *on average* you will be getting this event every 0.5 seconds.

There is also a `sys.timercount`^[223] read-only property. Timer counter is a free-running counter that is initialized to 0 when your device is powered up and increments every 0.5 seconds. The `sys.timercount` is useful in determining elapsed time, for instance, when you are waiting for something.

Here is an example. Supposing, you are supposed to wait for the serial data to arrive, but you are not willing to wait more than 10 seconds:

```
dim w as word
...
...
w=sys.timecount 'memorize time count at the beginning of waiting for
serial data
while ser.rxlen=0
    'nope, no data yet, do we still have time?
    if sys.timecount-w>20 then goto enough_waiting 'we quit after 10
seconds
    doevents 'polite waiting includes this
wend

...
...
```



The above example does not represent best coding style. Generally speaking, this is not a great programming but sometimes you just have to wait in a cycle.

PLL Management

PLL is a module of the device that transforms the clock generated by onboard crystal into higher frequency (x8 of the base for Tibbo devices). When the PLL is on, the device runs at 8 times the base frequency, when the PLL is off, the device runs at a "native" frequency of the crystal. Naturally, the device is 8 times faster (and consumes almost as much more power) when the PLL is on.

Not all Tibbo devices have PLLs- check "Platform-dependent Programming Information" topic inside your platform specifications section to find out if there is a PLL on your platform.

When a certain platform supports PLL, it will have a `sys.currentpll`^[218] read-only property and `sys.newpll`^[219] method to control the PLL. Due to the nature of PLL operation it is impossible to switch it on and off while the CPU is executing the firmware. The PLL needs time to "stabilize" its output frequency and it is not safe to let this happen when the CPU is running. Instead, the PLL is toggled when the CPU is in the reset state.

To change PLL mode, request new state through the `sys.newpll` method, then self-reset the device through the `sys.reboot`^[221] method. After the reboot the device emerges from reset with new PLL state (and PLL frequency already stabilized).

Here is a code example that makes sure that your device is running with PLL off:

```

sub on_sys_init
    if sys.currentpll=YES then
        sys.newpll(OFF)
        sys.reboot
    end if
end sub

```

External resets- power-up and RST pin reset (reset button reset)- set the PLL to default state (typically ON). On some devices there is a hardware jumper that defines the post-external reset state of the PLL.



Notice that PLL mode affects other objects- for example, baudrates of [serial ports](#)^[224] (this is why there is a [ser.div9600](#)^[250] property) and frequency generated by the [beep](#)^[387] object.

Serial Number

Tibbo devices currently in production use newer flash memory ICs featuring security register. This is a 128-byte register that has two 64-byte fields. The first field is one-time programmable, and the second field is pre-programmed at the factory and contains a unique serial number.

[Sys.serialnum](#)^[221] R/O property returns the entire 128-byte serial number or an empty string if the flash IC is of older type and does not have security register. [Sys.setserialnum](#)^[222] method is used to set the one-time programmable 64-byte field of the register:

```

Dim s As String

s="SERIAL NUMBER: 0123456789" 'this is the serial number we want to set
s=StrGen(64-Len(s),"") + s 'pad the string -- it must be exactly 64
                           characters in length
x=Sys.setserialnum(s) 'set the field!
...

```



Be careful -- you can only set the programmable portion of the serial number **once**. There is no way to correct a mistake!

Miscellaneous

The [sys.version](#)^[223] property returns the version of the TiOS (firmware) running on your device.

The [sys.halt](#)^[219] method can be used to halt the execution of your program. For example, you can use this to halt the program when the Ethernet interface failure is detected:

```
...
if net.failure= YES then
    sys.halt
end if
...
```

The [sys.reboot](#)^[221] method causes the device to reboot. This may be needed, for instance, to change the mode of the [PLL](#)^[215].

The [sys.runmode](#)^[221] read-only property informs whether the device is running in the release or debug mode. This data can be used by the program to exhibit different behavior- for example report all errors in the debug mode and try to "manage on its own" in the release mode (for more info see [Two Modes of Target Execution](#)^[27]).

The [sys.resettype](#)^[222] read-only property tells you what caused the most recent reset experienced by your device.

Properties, Methods, Events

This section provides an alphabetical list of all properties, methods, and events of the sys object.

.Buffalloc Method

Function: Allocates buffer memory as previously requested by "buffrq" methods of individual objects (such as [ser.rxbuffrq](#)^[262]).

Syntax: **sys.buffalloc**

Returns: ---

See Also: [sys.totalbuffpages](#)^[223], [sys.freebuffpages](#)^[218]

Details

Call this method after requesting all buffers you need through methods like [ser.txbuffrq](#)^[265] and [sock.cmdbuffrq](#)^[329].

This method takes significant amount of time (100s of milliseconds) to execute, during which time the device cannot receive network packets, serial data, etc. For certain interfaces like serial ports some incoming data could be lost.

Buffer (re)allocation for a specific object will only work if the corresponding object or part of the object to which this buffer belongs is idle. "Part" refers to a particular serial port of the [ser](#)^[224] object, or particular socket of the [sock](#)^[274] object, etc. to

which the buffer you are trying to change belongs. "Idle" means different things for different objects: [ser.enabled](#)^[25]= 0- NO for the serial port, [sock.statesimple](#)^[358]= 0- PL_SSTS_CLOSED for the socket, etc.

.Currentpll R/O Property (Selected Platforms Only)

Function:	Returns current PLL mode of the device
Type:	Enum (no_yes, byte)
Value Range:	0- NO: PLL is off, the device runs at low speed with reduced power consumption. 1- YES: PLL is on, the device runs at maximum speed, x8 faster than low speed.
See Also:	sys.newpll ^[219]

Details

After the external reset the device typically boots with PLL on. You can switch PLL off and on programmatically by using the [sys.newpll](#)^[219] method and then "self-resetting" the device using the [sys.reboot](#)^[221] method.

Actual PLL mode change only takes place after you "self-reset" the device using [sys.reboot](#)^[221] method or the device self-resets due to some other reason (for instance, there is a self reset after a new BASIC application upload, or when you hit "restart" button in TIDE). External resets- power-up and RST pin reset (reset button reset)- set the PLL to default state (typically ON). On some devices there is a hardware jumper that defines the post-external reset state of the PLL.

Not all Tibbo devices have PLL- check "Platform-dependent Programming Information" topic inside your platform specifications section to find out if there is a PLL on your platform.

.Freebuffpages R/O Property

Function:	Returns the number of free (not yet allocated) buffer pages (one page= 256 bytes).
Type:	Byte
Value Range:	0-255
See Also:	sys.totalbuffpages ^[223]

Details

Only changes after the [sys.buffalloc](#)^[217] method is used. "Preparatory" methods like [ser.rxbuffrq](#)^[262] do not influence what this property returns.

.Halt Method

Function:	Stops your program execution (halts VM).
Syntax:	sys.halt
Returns:	---
See Also:	sys.reboot ^[221]

Details

Causes the same result as when you press PAUSE in [TIDE](#)^[15] during the debug session. Once this method has been used, there is no way for your device to resume execution without your help.

.Newpll Method (Selected Platforms Only)

Function:	Sets new state of the PLL.
Syntax:	sys.newpll(newpllstate as off_on)
Returns:	---
See Also:	sys.currentpll ^[218]

Part	Description
newpllstate	Specifies what state the PLL will be in after the device emerges from internal reset: 0- OFF: PLL will be off, the device will emerge from reset at low speed with reduced power consumption 1- ON: PLL will be on, the device will emerge from reset at maximum speed, x8 faster than low speed

Details

Actual PLL mode change only takes place after you "self-reset" the device using [sys.reboot](#)^[221] method or the device self-resets due to some other reason (for instance, there is a self reset after a new BASIC application upload, or when you hit "restart" button in TIDE). External resets- power-up and RST pin reset (reset button reset)- set the PLL to default state (typically ON). On some devices there is a hardware jumper that defines the post-external reset state of the PLL.

Not all Tibbo devices have PLL- check "Platform-dependent Programming Information" topic inside your platform specifications section to find out if there is a PLL on your platform.



Notice that PLL mode affects other objects- for example, baudrates of [serial ports](#)^[224] (this is why there is a [ser.div9600](#)^[250] property) and frequency generated by the [beep](#)^[387] object.

On_sys_init Event

Function:	First event to be generated when your device boots up.
Declaration:	on_sys_init
See Also:	---

Details

Typically, initialization code for your application is placed here.

On_sys_timer Event

Function:	Periodic event.
Declaration:	on_sys_timer
See Also:	sys.timercount ^[223]

Details

Multiple on_sys_timer events may be waiting in the event queue. On_sys_timer event is not generated when the program execution is PAUSED (in [debug mode](#)^[27]).

New in **V1.2**. If the platform does not support the [sys.onsystimerperiod](#)^[220] property, then the interval of this event generation is fixed at 0.5 seconds. If the platform does support this property, then the period is adjustable.

Onsystimerperiod Property (Selected Platforms Only)

Function:	New in V1.2 . Sets/returns the period for the on_sys_timer ^[220] event generation expressed in 10ms intervals.
Type:	Byte
Value Range:	0-255. Default = 50 (500ms).
See Also:	---

Details

Defines, in 10ms increments, the period at which the [on_sys_timer](#)^[220] event will be generated. Platforms that do not support this property have the period fixed at 0.5 seconds.

.Reboot Method

Function:	Causes your device to reboot.
Syntax:	sys.reboot
Returns:	---
See Also:	sys.currentpll ^[218] , sys.runmode ^[221] , sys.resettype ^[222] , and sys.halt ^[219]

Details

After the device reboots it will behave as after any other reboot: enter PAUSE mode if your program was compiled for debugging, or start execution if the program was compiled for release (see [Two Modes of Target Execution](#)^[27]).

The PLL mode will change after the reboot if you requested the changed through [sys.newpll](#)^[219] method.

.Runmode R/O Property

Function:	Returns current run (execution) mode.
Type:	Enum (pl_sys_mode, byte)
Value Range:	0- PL_SYS_MODE_RELEASE: debugging is not possible, application execution starts immediately after device powers up. Severe errors such as "division by zero" are ignored and do not stop execution. 1- PL_SYS_MODE_DEBUG: debug mode in which it is possible to cross-debug the application (under the control of TIDE ^[15] software). Application execution is not started automatically after the power up. Severe errors such as "division by zero" halt execution.
See Also:	sys.currentpll ^[218] , sys.newpll ^[219] , sys.resettype ^[222] , and sys.halt ^[219]

Details

For some programs, it may be useful to know if the program is currently executing in Debug Mode or Release Mode(see [Two Modes of Target Execution](#)^[27]).

Serialnum R/O Property

Function:	Returns the 128-byte string containing the serial number of the device.
Type:	String
Value Range:	A string of 128 bytes.
See Also:	Serial Number ^[216]

Details

The serial number comes from the security register of the flash IC. Older generation of flash ICs used in Tibbo devices did not have the security register. This property will return an empty string if the security register is not present. First 64 bytes of the security register are preprogrammed with a serial number, and remaining 64 bytes are one-time programmable. Use the [sys.setserialnum](#)^[222] method to set the data.

Setserialnum Method

Function:	Sets the programmable portion (64 bytes) of the device's 128-byte serial number.
Syntax:	sys.setserialnum(byref str as string) as ok_ng
Returns:	0- OK: The serial number was set successfully. 1- NG: Serial number programming failed.
See Also:	Serial Number ^[222]

Details

The serial number is stored in the security register of the flash IC. Older generation of flash ICs used in Tibbo devices did not have the security register. This method will return 1- NG if you attempt to set the serial number of the device that does not have the security register.

For the method to work, the input string must be exactly 64 bytes in length, otherwise 1- NG will be returned. The security register can only be programmed once. Attempting to program it again will fail (again, with 1- NG code).

Note that using this method disrupts the operation of the flash memory. The operation uses buffer 1 of the flash IC for temporary data storage, so invoking this method will alter the buffer contents. To prevent potential data errors, invoking the method sets [fd.ready](#)^[476]= 0- NO automatically.

The Entire 128-byte serial number can be obtained through the [sys.serialnum](#)^[221] R/O property.

.Resettype R/O Property

Function:	Returns the type of the most recent hardware reset.
Type:	Enum (pl_sys_reset_type, byte)
Value Range:	0- PL_SYS_RESET_TYPE_INTERNAL: The most recent reset was generated internally. 1- PL_SYS_RESET_TYPE_EXTERNAL: The most recent reset was generated externally.
See Also:	sys.currentpll ^[218] , sys.runmode ^[221] , sys.reboot ^[221] , and sys.halt ^[219]

Details

Internal resets are generated when the device self-reboots. This can be caused by the execution of [sys.reboot](#)^[221] in your application, or command from TIDE.

External resets are the ones that are caused by power-cycling (turning the device off and back on) or applying a reset pulse to the RTS line of the device (pushing reset button).

.Timercount R/O Property

Function:	Returns the time (in half-second intervals) elapsed since the device powered up.
Type:	Word
Value Range:	0-65535
See Also:	on_sys_timer ^[220]

Details

Once this timer reaches 65535 it rolls over to 0.

.Totalbuffpages R/O Property

Function:	Returns the total amount of memory pages available for buffers (one page= 256 bytes).
Type:	Byte
Value Range:	0-255
See Also:	sys.buffalloc ^[217] , sys.freebuffpages ^[218]

Details

Calculated as total available variable memory (RAM) in pages minus number of pages required to store variables of the current project.

.Version R/O Property

Function:	Returns firmware (TiOS) version string.
Type:	String
Value Range:	Whatever is set in firmware, for example "<EM202-1.00.00>"
See Also:	---

Details

Ser Object



This is the *serial port* object. This object encompasses all serial ports available on a particular platform. Each serial port can function as a standard [UART](#)^[226], [Wiegand](#)^[229] device, or [clock/data](#)^[232] device. Direct support for Wiegand and clock/data interfaces is a unique feature of the serial port object.

Each serial port has a set of standard programmable parameters that you would expect to find, such as baudrate and parity. In addition, it has some unique features like the ability to automatically filter out so-called 'escape sequences' or select UART, Wiegand, or clock/data operating mode.

Follows is the list of features offered by the serial port object:

- Ability to work in UART, Wiegand, or clock/data mode
- Fully asynchronous operation with separate "data arrival" and "data sent" events.
- Automatic data overrun detection on the RX buffer.
- Adjustable receive (RX) and transmit (TX) buffer sizes for optimal RAM utilization.
- Data "grouping" in the RX buffer based on "intercharacter" delay (gap between two consecutive arriving characters) and amount of data in the buffer.
- Optional automatic port disabling when a data group has been received.
- Buffer shorting feature for fast data exchange between the ser object and other objects (such as the [sock](#)^[274] object) that support standard Tibbo Basic data buffers.

For the UART mode:

- Ability to set any baudrate (that is physically possible on a particular platform) by specifying "divider value" instead of a "baudrate from a list" as is usually done on other products
- Standard parity selection: parity off, even, odd, mark, or space.
- Choice of word length- 7 or 8 data bits/word.
- Full duplex or half duplex operation.
- Optional automatic CTS/RTS flow control in the full-duplex mode.
- Automatic direction control via RTS line in the half-duplex mode with direction control polarity selection.
- Automatic detection of "escape sequences", active even when the buffer shorting is enabled (see below).

What's new in V1.1

Enter topic text here.

Overview 3.2.2

This section covers the serial port object in detail. Here you will find:

- [Anatomy of a Serial Port](#)^[225]
- [Three modes of the serial port](#)^[225]
- [Port Selection](#)^[234]
- [Serial Settings](#)^[236]
- [Sending and Receiving Data \(TX and RX buffers\)](#)^[239]

Anatomy of a Serial Port

A serial port is composed of actual hardware which controls serial port lines, and of buffers that store incoming data (which is to be processed by your application) and outgoing data (which has not yet left the port).

The serial port object contains properties, methods and events which relate both to the buffers and the UART itself (see [Serial Settings](#)^[236]).

The *buffers* available are:

- The **TX buffer**, which contains data due to be sent out of the port (i.e., it's the transmit of *your device!*). Your Tibbo Basic application puts the data into the TX buffer.
- The **RX buffer**, which contains incoming data received by the port. This data is to be processed by your application.

The *logical lines* available are:

- The **TX/W1out/dout output** line.
- The **RX/W1in/din input** line.
- The **RTS/W0out/cout output** line.
- The **CTS/W0&1in/cin input** line.

TX/W1out/dout and RX/W1in/din lines always have fixed "position" in the device i.e. they cannot be re-mapped to a different I/O pin. RTS/W0out/cout and CTS/W0&1in/cin lines can be re-mapped on select devices. Also, depending on the device and the serial port mode you may or may not require to explicitly configure the lines of the serial port as inputs or outputs. Sometimes it will happen automatically, and sometimes you need to take care of this in your application through the [io](#)^[365] object. You will find this information in the "Platform-related Programming Information" topic inside your platform specifications section.

Details of I/O line usage in each of the three operating modes of the port can be found [here](#)^[225].

Three Modes of the Serial Port

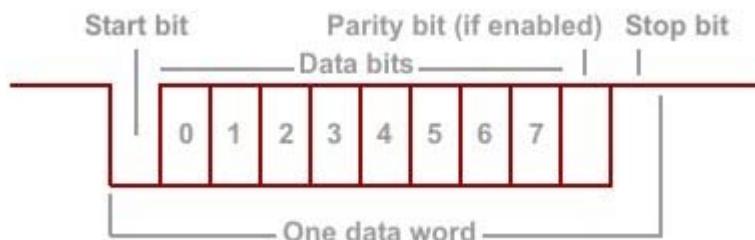
Each port of the serial object can operate in one of the three modes:

- [UART Mode](#)^[226]
- [Wiegand Mode](#)^[229]
- [Clock/data Mode](#)^[232]

The following subsections detail the signals that the serial port sends and expects to receive in each mode.

UART Mode

In the UART mode the serial port has standard UART functionality. You can select the baudrate, parity, number of bits in each character, and full- or half-duplex operation (see [Serial Settings](#)^[236]). The UART works with signals of "TTL-serial" polarity -- RX and TX lines are at logical HIGH when no data transmission is taking place, the start bit is LOW, stop bit is HIGH (shown below for the case of 8 bits/character and enabled parity).



UART data is sent and received via **TX** output and **RX** input lines. Two additional lines- **RTS** output and **CTS** input may also be used depending on the serial port setup (see below).



Please, remember that on your platform you may be required to correctly configure some of your serial port's lines as inputs or outputs through the [io.enabled](#)^[370] property of the [io](#)^[365] object. Additionally, you may have the freedom of re-mapping certain serial port lines to different I/O pins of the device if required. See "Platform-dependent Programming Information" topic inside your platform specifications section.

How the serial port sends and receives UART data

The serial port can send and receive the data with no parity, or even, odd, mark, or space parity configuration. Additionally, you can specify the number of bits in each character (7 or 8). The serial port takes care of parity calculation automatically. When parity bit is enabled, it will automatically calculate parity bit value for each character it transmits. When receiving, the serial port will correctly process incoming characters basing on the specified number of bits and parity mode. Actual parity check is not done. The serial port will receive the parity bit but won't actually check if it is correct.



Actual parity check is not done. The serial port will receive the parity bit but won't actually check if it is correct. Parity is mostly kept for compatibility with older devices, so the serial port transmits it correctly.

How the UART data is stored in the RX and TX buffers of the serial port

When in the UART mode, each data byte in the TX or RX buffer of the serial port represents one character. When the serial port is configured to send and receive 7-bit characters, the most significant bit of each byte in the RX buffer will be 0, and the most significant bit of each byte in the TX buffer will be ignored. Parity bits are not stored in the buffers. For the outgoing data stream, the serial port will calculate and append the parity bit automatically. For the incoming serial data, the

serial port will discard the parity bit so the RX buffer will only get "pure" data.

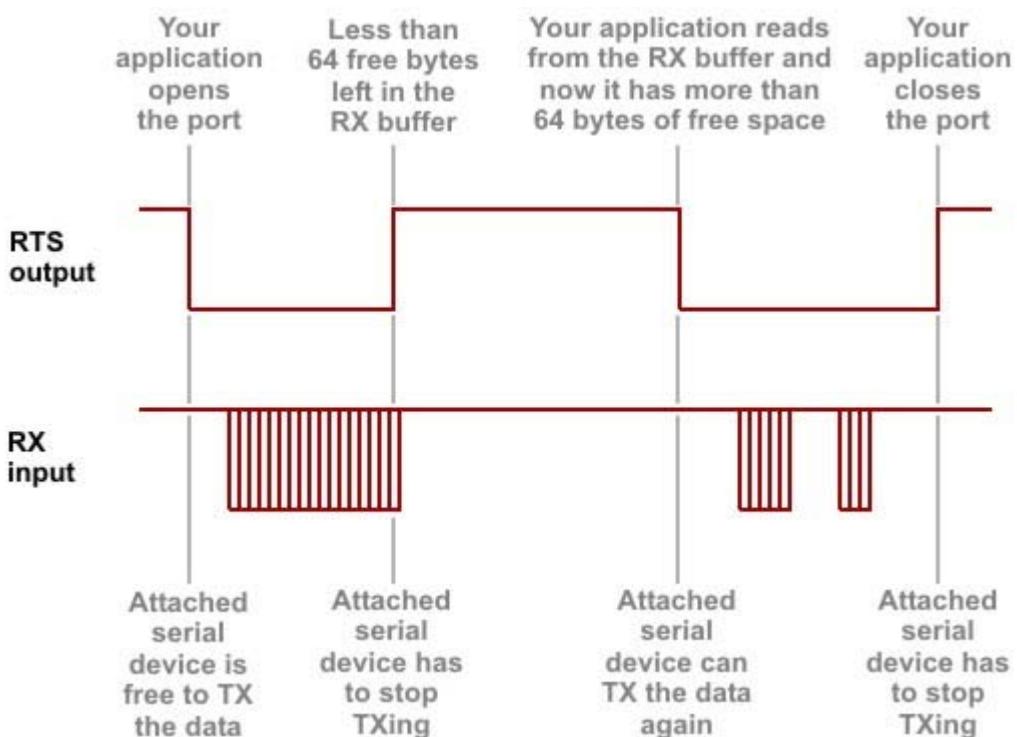
Full-duplex operation

The full-duplex mode of operation is suitable for communicating with RS232, RS422 devices, 4-line RS485 devices, and most TTL-serial devices. Naturally, external transceiver IC is needed for RS232, RS4222, or RS485. TTL serial devices can be connected to the serial port lines directly in most cases.

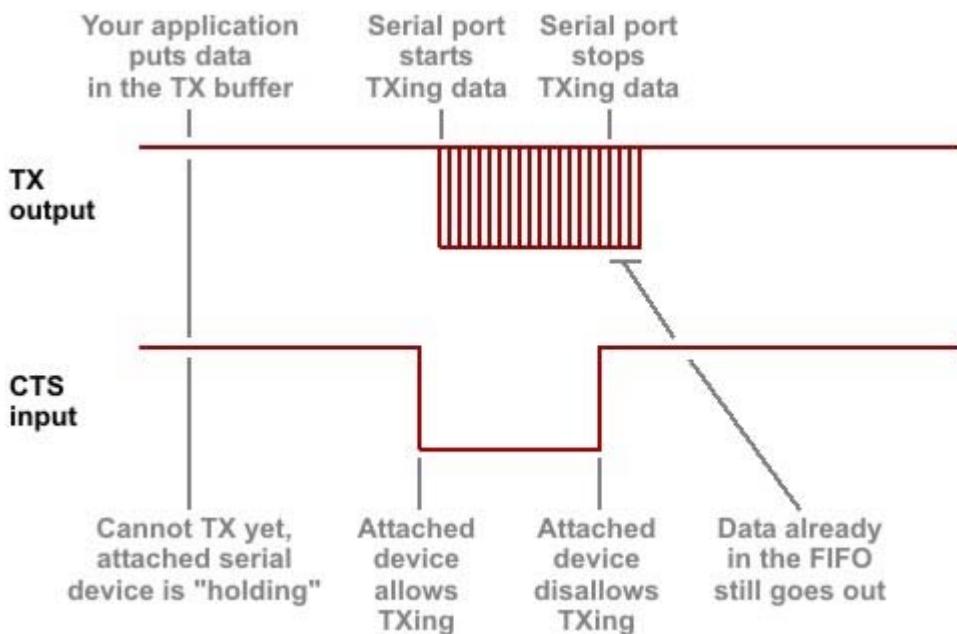
In the full-duplex mode, the RTS output and CTS input lines can be used for optional flow control. The RTS output is low whenever the serial port is ready to receive the data from "attached" serial device (port is opened and the RX buffer has at least 64 bytes of free space). The RTS line is high whenever the serial port is closed or the RX buffer has less than 64 bytes of free space left. Figure below illustrates RTS operation.



All diagrams show TTL-serial signals. If you are dealing with the lines of the RS232 port you will see all signals in reverse!



The CTS input is used by the serial port to check if attached serial device is able to accept the data. The serial port will only start to send the data when the CTS input is low. The serial port will stop sending the data once the line goes high. Note, that some Tibbo devices have a hardware buffer called "FIFO" ("first-in-first-out" if you really need to know ;-). Once the TX data is in the FIFO it will be sent out even if the CTS line goes low. Therefore, after the attached serial device switches the CTS line to low the serial port may still output the number of bytes not exceeding the capacity of the FIFO. See "Platform-dependent Programming Information" topic inside your platform specifications section for the serial port FIFO size on your platform.



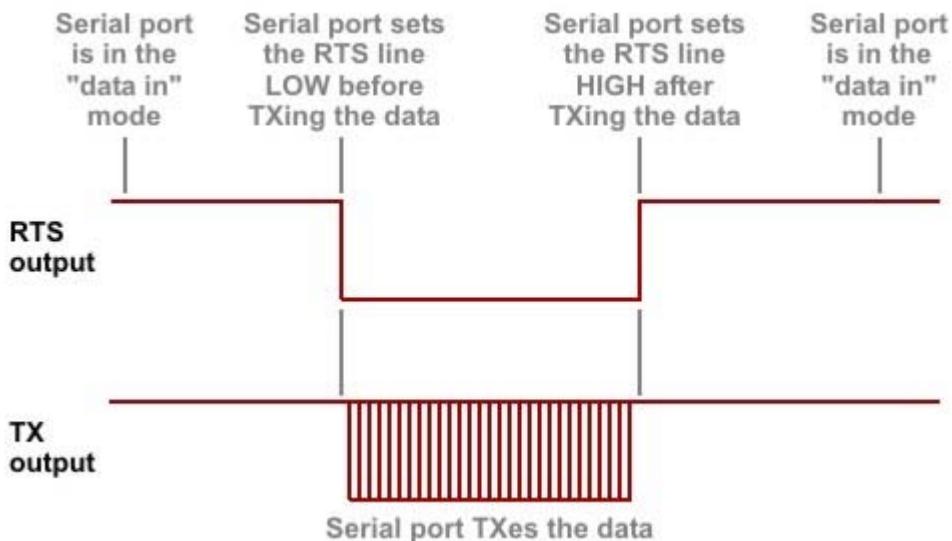
When the flow control is enabled, the RTS and CTS lines are controlled by the serial port and cannot be manipulated by your application. When the flow control is off, your application can set and read the state of these lines through the [io³⁶⁵](#) object.

Half-duplex operation

The half-duplex mode of operation is suitable for communicating with 2-wire RS485 devices. Again, appropriate interface transceiver IC is required to be connected to the serial port.

In the half-duplex mode the RTS output is used to control data transmission direction. You can select the polarity of the direction control signal, i.e. which state will serve as "data in" direction, and which- as "data out" direction (see [Serial Settings](#)^[236]).

When the serial port has no data to transmit (the TX buffer is empty), it is always ready to receive the data, so the RTS line is in the "data in" state. When the serial port needs to send out some data (the TX buffer is not empty) it switches the RTS line into the "data out" state, transmits the data, then switches the RTS back into the "data in" state. Assuming "LOWFORINPUT" direction control polarity, direction control looks like this:



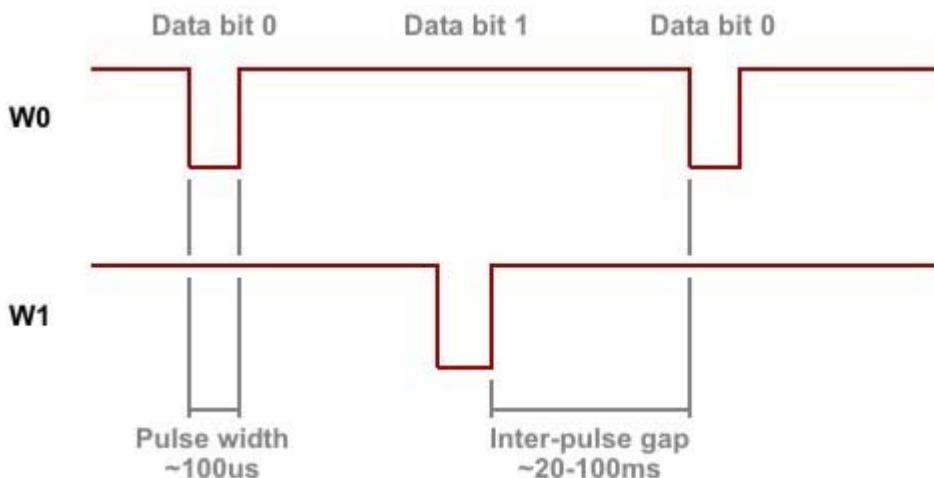
The CTS line is not controlled by the serial port when in the half-duplex mode. Your application can manipulate this line through the [io\[365\]](#) object.

Wiegand Mode

In the Wiegand mode the serial port is able to receive the data directly from any Wiegand device, such as card reader and also output the data in the Wiegand format, as if it was a card reader itself. Wiegand interface is popular in the security, access control, and automation industry.

Standard Wiegand data transmission is shown below. There are two data lines- W0 and W1. Negative pulse on the W0 line represents data bit 0. Negative pulse on the W1 line represents data bit 1. There is no standard Wiegand timing, so pulse widths as well as inter-gap widths vary greatly between devices. Averagely, pulse width is usually in the vicinity of 100 μ s (microseconds), while the inter-pulse gap is usually around 20-100ms (milliseconds).

There is no explicit way to indicate the end of transmission. Receiving device either counts received bits (if it knows how many to expect) or assumes the transmission to be over when the time since the last pulse on the W0 or W1 line exceeds certain threshold, for example, ten times the expected inter-pulse gap.



The serial port outputs Wiegand data through **W0out** and **W1out** lines and receives the data via **W0&1in** and **W1in** lines. "W0&1in" means that the signal on this input must be a logical AND of W0 and W1 output lines of attached Wiegand device (see below for details). Your application *should not* attempt to work with W0out and W1out outputs directly through the [io](#)^[365] object when the serial port is in the Wiegand mode.



Please, remember that on your platform you may be required to correctly configure some of your serial port's lines as inputs or outputs through the [io.enabled](#)^[370] property of the [io](#)^[365] object. Additionally, you may have the freedom of re-mapping certain serial port lines to different I/O pins of the device if required. See "Platform-dependent Programming Information" topic inside your platform specifications section.

How the serial port sends and receives raw Wiegand data

There are many Wiegand data formats currently in use. These formats define how "raw" data bits are processed and converted into actual data. Typically, there are 2 parity bits- one at the beginning, and another one at the end of Wiegand data. Parity calculation, however, varies from format to format. Additionally, the length of Wiegand output is not standardized.

All this makes it impossible for the serial port object to verify incoming Wiegand data, i.e. check the data length and calculate the checksum. Instead, this task is delegated to your application while the serial object only receives raw data. Similarly, before sending out Wiegand data your application needs to prepare this data in the desired format- the serial object itself will output any data stream.

How the Wiegand data is stored in the RX and TX buffers of the serial port

When in the Wiegand mode, each data byte in the TX or RX buffer of the serial port represents one bit of Wiegand data. This bit is recorded in the least significant bit position of each data byte in the buffer. For your application's convenience, when the serial port receives Wiegand bit stream, it adds an offset of 30Hex to each data bit. Therefore, the data recorded into the RX buffer can only consist of bytes 30H and 31H. These correspond to ASCII characters '0' and '1'. This way, when your application reads RX buffer contents into a string variable the data will be "readable" without any additional conversion (ASCII characters with codes 0 and 1 would not be "readable").

When the serial port outputs Wiegand data, it only takes bit 0 of each byte in the TX buffer. Other bits can contain any data. You can, for instance, put a string of ASCII characters '0' and '1' into the TX buffer and these will be correctly interpreted as data bits 0 and 1. This, again, is convenient for your BASIC application.

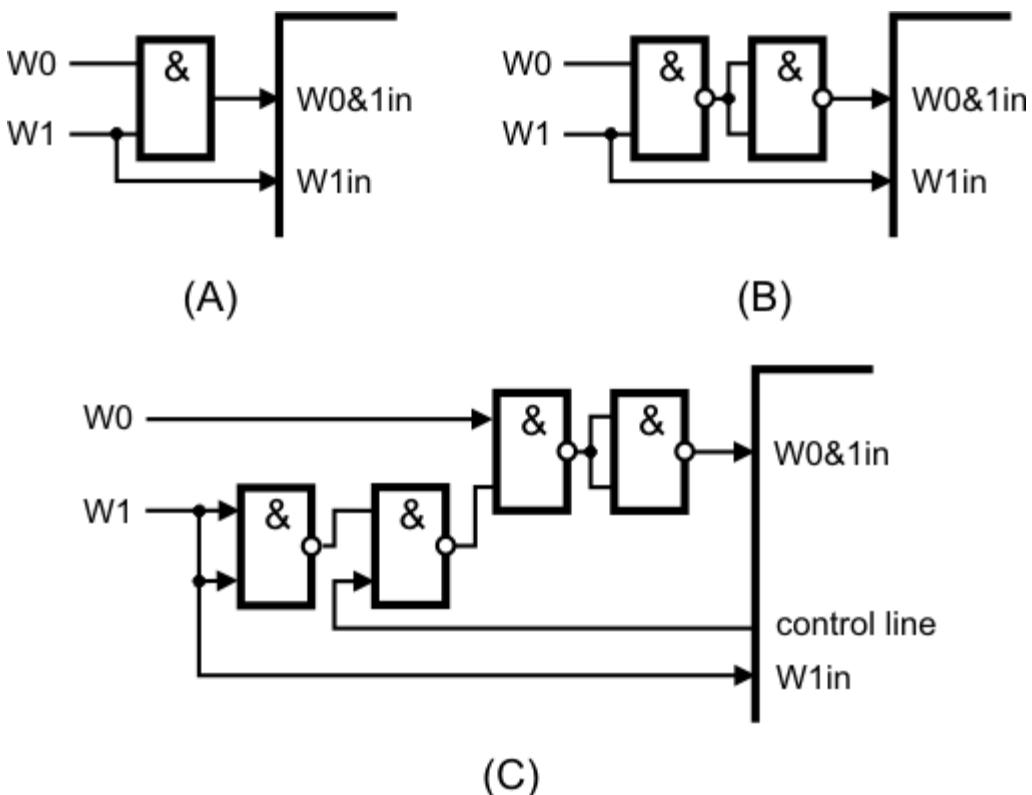
How the serial port transmits Wiegand data

Wiegand data output timing is fixed and your application cannot change it. Data pulses are 100uS wide and inter-pulse gaps are 20mS wide.

How the serial port receives Wiegand data

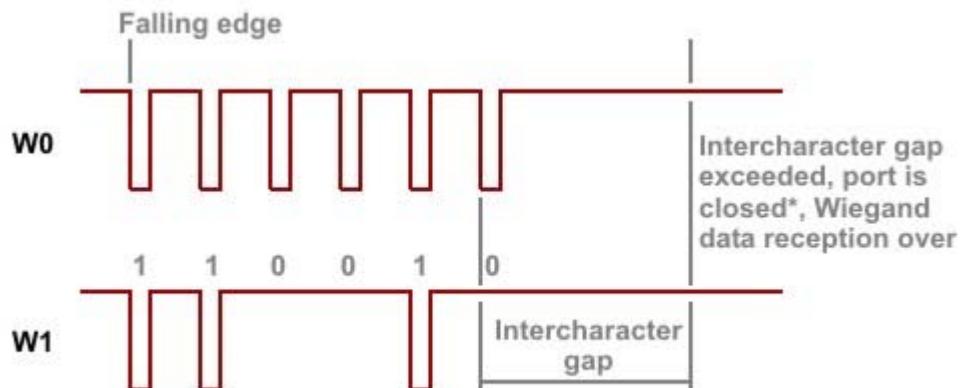
You already know that W0&1in input of the serial port must receive a logical AND of W0 and W1 output of attached Wiegand device. A simple AND gate will do the job (figure A below). Actually, NOR-AND gates are more popular and these can be

used too (figure B). In case you are building a product that will also accept [clock/data](#)^[232] input, you may need to control whether the W0&1in input should receive a logical AND of two lines, or just one of the lines. Schematic diagram C uses an additional I/O line of the device to control this. When the control line is HIGH the W0&1in input receives a logical AND of both W0 and W1 lines, when the control line is LOW, the W0&1in input receives just the signal from the W0 line. Four gates are required for this, so you will get away with using a single 74HC00 IC.



The serial port does not require an incoming Wiegand data stream to adhere to any strict timing. The port is simply registering high-to-low transitions on the W0&1in line. When such transition is detected, the port checks the state of W1 line. If the line is HIGH, data bit 0 is registered, when the line is low, data bit 1 is registered.

The end of Wiegand transmission is identified by timeout- the serial port has a special property for that, called "intercharacter delay" (see [Serial Settings](#)^[233]). Another property- "auto-close"- can be used to disable the serial port after the delay has been encountered. This way, when the Wiegand output is over the port will be disabled and no further data will enter the port until you re-enable it.



**If the port is configured to auto-close after the gap*

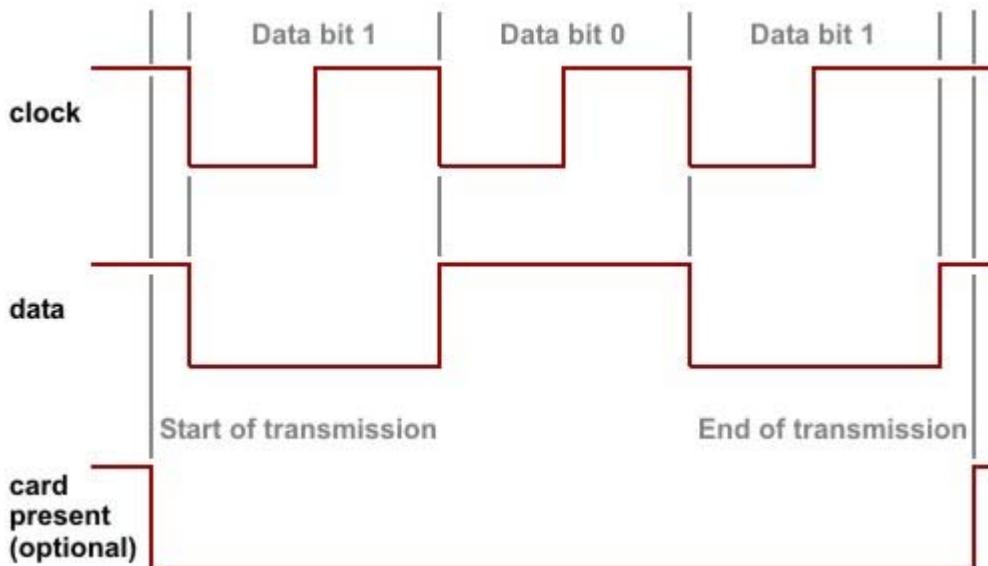
Clock/Data Mode

In the clock/data mode the serial port is able to receive the data directly from any clock/data or "magstripe interface" device, such as a card reader. The serial port is also able to output the data in the clock/data format, as if it was the card reader itself. Magstripe and clock/data interfaces are popular in the security, access control, automation, and banking industry.

Standard clock/data transmission is shown below. There are two data lines- clock and data. Each negative transition on the clock line marks the beginning of the data bit. The data line carries actual data. When the state of the data line is LOW it means data bit 1, and vice versa. There is no standard clock/data timing and some devices, such as non-motorized magnetic card readers, output the data at variable speeds (depending on how fast the user actually swipes the card).

The magstripe interface only differs from the clock/data interface in that it has a third line- card present. This line goes LOW before the data transmission and goes back to HIGH after the transmission is over. The serial port does not require the card present line for data reception. Just like with [Wiegand data](#)^[229], it identifies the end of incoming data by measuring the time since the last negative transition on the clock line. For data transmission, your application can easily use any regular I/O line to serve as card present line.

Compared to Wiegand interface, the data format of clock/data interface is very standardized and its varieties include standard data formats for different "tracks" of the magnetic card. Most clock/data devices you will actually encounter have nothing to do with magnetic cards but terminology persists.



The serial port outputs clock/data signals through **cout** and **dout** lines and receives the data via **cin** and **din** lines. Your application *should not* attempt to work with cout and dout outputs directly through the [io](#)^[365] object when the serial port is in the clock/data mode.



Please, remember that on your platform you may be required to correctly configure some of your serial port's lines as inputs or outputs through the [io.enabled](#)^[370] property of the [io](#)^[365] object. Additionally, you may have the freedom of re-mapping certain serial port lines to different I/O pins of the device if required. See "Platform-dependent Programming Information" topic inside your platform specifications section.

How the serial port sends and receives raw clock/data data

Clock/data from different "tracks" has different encoding. Encoding defines how "raw" data bits are processed and converted into actual data. To allow maximum flexibility, and also to maintain the data processing style used by the [Wiegand interface](#)^[229], the serial port leaves the task of converting between the raw and actual data to your application. The serial port only sends and receives raw data without checking or transforming its contents.

How the clock/data stream is stored in the RX and TX buffers of the serial port

When in the clock/data mode, each data byte in the TX or RX buffer of the serial port represents one bit of the clock/data stream. This bit is recorded in the least significant bit position of each data byte in the buffer. For your application's convenience, when the serial port receives clock/data bit stream, it adds an offset of 30Hex to each data bit. Therefore, the data recorded into the RX buffer can only consist of bytes 30H and 31H. These correspond to ASCII characters '0' and '1'. This way, when your application reads RX buffer contents into a string variable the data will be "readable" without any additional conversion (ASCII characters with codes 0 and 1 would not be "readable").

When the serial port outputs clock/data stream, it only takes bit 0 of each byte in

the TX buffer. Other bits can contain any data. You can, for instance, put a string of ASCII characters '0' and '1' into the TX buffer and these will be correctly interpreted as data bits 0 and 1. This, again, is convenient for your BASIC application.

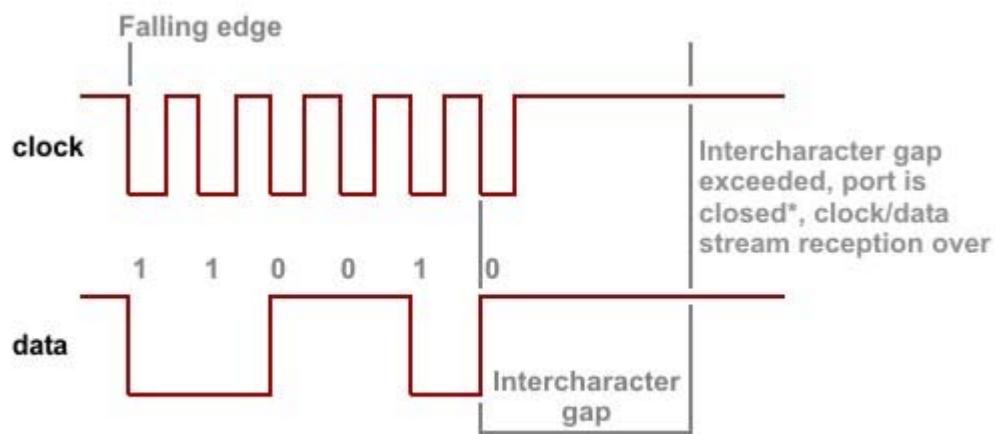
How the serial port transmits clock/data stream

Clock/data output timing is fixed and your application cannot change it. The data is output at a rate of 400us/bit (both LOW and HIGH phases of the clock signal are 200us in length).

How the serial port receives clock/data stream

The serial port does not require an incoming clock/data stream to adhere to any strict timing. The port is simply registering high-to-low transitions on the clock line. When such transition is detected, the port checks the state of the data line. If the line is HIGH, data bit 0 is registered, when the line is low, data bit 1 is registered.

The end of clock/data transmission is identified by timeout- the serial port has a special property for that, called "intercharacter delay" (see [Serial Settings](#)^[236]). Another property- "auto-close"- can be used to disable the serial port after the delay has been encountered. This way, when the clock/data output is over the port will be disabled and no further data will enter the port until you re-enable it.



**If the port is configured to auto-close after the gap*

Port Selection

There may be platforms with more than one serial port. You can obtain the number of serial ports available for your platform using the [ser.numofports](#)^[258] property.

Since there can be multiple ports, you must state which port you are referring to when changing properties or invoking methods. This is done using the [ser.num](#)^[257] property. For example:

```
ser.mode = PL_SER_MODE_WIEGAND
```

Can you tell what serial port the statement above applies to? Neither can the platform. Thus, the correct syntax would be:

```
ser.num = 0  
ser.mode = PL_SER_MODE_WIEGAND
```

Now the platform knows what port you're working with. Once you have set the *port selector* (using ser.num), every method and property after that point is taken to refer to that port. Thus:

```
ser.num = 0  
ser.enabled = 1  
ser.baudrate = 1  
ser.bits = 1 ' etc
```

The events generated by the ser object are not separate for each port. An event such as [on_ser_data_arrival](#)^[258] serves all serial ports on your platform. Thus, when an event handler for the serial port object is entered, the port selector is automatically switched to the port number on which the event has occurred:

```
sub on_ser_data_arrival  
  
    dim s as string  
    s = ser.getdata(255) ' Note that you did not have use ser.num before  
    this statement.  
  
end sub
```

As a result of this automatic switching, when an event handler for a serial port event terminates, the ser.num property retains its new value (nothing changes it back). You must take this into account when processing other event handlers which make use of the serial port (and are not serial port events). In other words, you should explicitly set the ser.num property whenever entering such an event handler, because the property might have been automatically changed prior to this event. To illustrate:

```

sub on_sys_init ' This is always the first event executed.

ser.num = 0 ' Supposedly, this would make all subsequent properties and
methods refer to this port.

end sub

sub on_ser_data_arrival ' Then, supposing this event executes.

dim s as string
s = ser.getdata(255) ' However, this event happens on the second port. So
now ser.num = 1.

end sub

sub on_sock_data_arrival ' And then this socket event executes.

ser.txclear ' You meant to do this for ser.num = 0 (as specified at
on_sys_init). But now port.num was changed to 1. You did not explicitly
specify a ser.num here, and now the tx buffer for the wrong port is
cleared. Oops.

end sub

```

Same precautions should be taken when using [doevents](#)^[68]. This is because doevents will let other events execute and so serial object events will potentially execute and cause the ser.num to change.

To recap, only one of two things may change the current ser.num: **(1)** Manual change or **(2)** a serial port event. And you cannot assume the number has remained unchanged if you set it somewhere else (because a serial port event might have happened since).



Specifying a port number for a single-port platform may seem redundant, but it makes your program portable. You will have an easier time migrating your program to a multi-port platform in the future.

Serial Settings

This topic briefly outlines the range of configurable options available on the serial port (this does not include the bulk of data on RX and TX buffers, which are described [here](#)^[239]).

Opening and closing the serial port

The [ser.enabled](#)^[251] property defines whether the port is opened or closed. The port is closed by default, so you need to open it explicitly. The serial port won't receive or transmit the data when it is closed, but you will still be able to access its RX and TX buffers even at that time.

Serial port lines remapping

Depending on your platform, you may or may not be allowed to remap **RTS/W0out/cout** output and **CTS/W0&1in/cin** input, i.e. choose what I/O pins of the device these lines should be on. See "Platform-dependent Programming Information" topic inside your platform specifications section. If your device supports remapping, you can use [ser.ctsmap](#)^[249] and [ser.rtsmap](#)^[261] properties to

select required mapping. If your device does not support remapping then its platform will not have these properties. You can only perform remapping when the serial port is closed.

UART, Wiegand, or clock/data mode selection

The [ser.mode](#)^[255] property selects [UART](#)^[226], [Wiegand](#)^[229], or [clock/data](#)^[232] mode for the serial port. You can only change the mode when the serial port is closed.

How the incoming data is committed

One important concept you will need to understand is "RX data committing". You already know that the serial port records all incoming data into the RX buffer. New information is that the data recorded into the buffer is not immediately "reported" to your application. Instead, this buffer remains "uncommitted" until certain conditions are met. Uncommitted data is effectively invisible to your application, as if it is not there at all.

For [UART](#)^[226] mode, the data is committed either when the total amount of uncommitted and committed data in the RX buffer exceeds 1/2 of this buffer's capacity or when the intercharacter gap, defined by the [ser.interchardelay](#)^[254] property is exceeded. For [Wiegand](#)^[229] and [clock/data](#)^[232] modes, the first condition is not monitored, so only the intercharacter gap can commit the data.

The intercharacter gap is the time elapsed since the start of the most recent UART character reception in the UART mode or the most recent falling edge on the W0&1in/cin line in the Wiegand and clock/data modes. The idea is that once the data stops coming in, the serial port starts counting the delay. Once the delay exceeds the time set by the [ser.interchardelay](#) property, the data is committed and becomes visible to your application.

Another property -- [ser.autoclose](#)^[248] -- defines whether the port will be closed ([ser.enabled](#)^[251]= NO) once the intercharacter gap reaches [ser.interchardelay](#) value.

For the UART mode, the intercharacter delay allows your application to process the data more efficiently. By keeping the data invisible for a while the serial port can accumulate a large chunk of data that your application will be able to process at once. Imagine, for instance, that the data is flowing into the serial port character by character and your application has to also process this stream character by character. The overhead may be significant and overall performance of your application greatly reduced! Now, if this incoming data is combined into sizeable portions your application won't have to handle it in small chunks, and this will improve performance. Of course, you need to strike a balance here -- attempting to combine the data into blocks that are too large may reduce your application's responsiveness and make your program appear sluggish.

Notice, that the intercharacter gap is not counted when the new data is not being received because the serial port has set the RTS line to LOW (not ready). This could happen when flow control is enabled (more on flow control below).

For the Wiegand and clock/data modes, the intercharacter delay is the way to detect the end of incoming data stream. You are recommended to program the gap to about 10 times the data rate. For example, if you are receiving the Wiegand data at a rate of 1 bit per 20ms, then set the delay to 200 ms and it will serve as a reliable indicator of transmission end. This is when the [ser.autoclose](#) will come handy- once the gap is detected the port will be closed and this will prevent another Wiegand transmission from entering the RX buffer before your application processes the previous one.

How the outgoing data is committed

Outgoing data uses similar "data committing" concept as the incoming data. The objective is to be able to commit the data for sending once, even if the data was prepared bit by bit. This way your application can avoid sending out data in small chunks. [Buffer Memory Status](#)^[240] topic details this.

UART mode settings

Several settings are unique to [UART](#)^[226] mode. The serial port has all the usual UART-related communication parameters: [baudrate](#)^[248], [parity](#)^[260], [7/8 bits](#)^[249]. There is no property to select the number of stop bits. Second stop bit can be emulated by setting `ser.parity= 3- PL_SER_PR_MARK`.

The baudrate property actually keeps a divisor value. You can set any baudrate you want by providing the correct divisor. There is even a read-only [ser.div9600](#)^[250] property that allows you to calculate the divisor value in a platform-independent way, by returning the required divisor value (for the current platform) to reach 9600bps.

The actual baudrate is calculated as follows: `baudrate=(9600*ser.div9600)/ser.baudrate`. For example, if we need to achieve 38400bps and `ser.div9600` returns 12, then we need to set `ser.baudrate=3`, because $(9600*12)/38400=3$. This serves as a platform-independent baudrate calculation, as `ser.div9600` will return different values for different platforms.

For example:

```
function set_baud(baud as integer) as integer
'baud: 0- 1200, 1-2400, 2-4800, 3- 9600, 4-19200, other-38400

select case baud
    case 0: ser.baudrate=ser.div9600*8 '9600/1200=8
    case 1: ser.baudrate=ser.div9600*4 '9600/2400=4
    case 2: ser.baudrate=ser.div9600*2 '9600/4800=2
    case 3: ser.baudrate=ser.div9600*8 '9600/9600=1
    case 4: ser.baudrate=ser.div9600/2 '19200/9600=2
    case else: ser.baudrate=ser.div9600/4 '38400/9600=4
end function
```

The serial port can be used in full-duplex or half-duplex mode, as determined by the [ser.interface](#)^[251] property (see [UART Mode](#)^[226] for details). In the full-duplex mode, the serial port can optionally use the [flow control](#)^[252]. In the half-duplex mode, you can select the [polarity](#)^[250] of the direction control signal.

In the UART mode, the serial port can recognize so-called 'escape sequences'. They are defined using the [ser.esctype](#)^[253] and [ser.escchar](#)^[254] properties.

When such a sequence is recognized, a special event ([on_ser_esc](#)^[255]) is generated and the port is disabled. Of course this can be achieved programmatically, but it would require you to parse all incoming data and really slow things down. Escape sequences were implemented with efficiency and speed in mind.

Escape sequences are rather arbitrary. They follow the style of escape sequences that Tibbo introduced before. However, they are useful for certain things. For example , if your application has a normal mode and serial 'setup' mode, you can use this sequence to switch into setup mode.

Escape sequence will work even if you are using [buffer shorting](#)^[245], and that makes them especially powerful. If you are building a device which just routes the data between a serial interface and an Ethernet interface, you will use buffer shorting for performance, but you could still detect the escape sequences to switch into the

serial programming mode or perform some other similar task.

Sending and Receiving Data (TX and RX buffers)

The serial port sends and receives data through TX (transmit) and RX (receive) buffers. Read on and you will know how to allocate memory for buffers, use them, handle overruns, and perform other tasks related to sending and receiving of data.

Allocating Memory for Buffers

Each buffer has a certain size, i.e., a memory capacity. This capacity is allocated upon request from your program. When the device initially boots, no memory is allocated to buffers at all.

Memory for buffers is allocated in pages. A *page* is 256 bytes of memory. Allocating memory for a buffer is a two-step process: First you have to request for a specific allocation (a number of pages) and then you have to perform the actual allocation. For the RX buffer, request memory using [ser.rxbuffrq](#)^[262], and for the TX buffer, request it using [ser.txbuffrq](#)^[265].

The allocation method ([sys.buffalloc](#)^[217]) applies to all buffers previously specified, in one fell swoop. Hence:

```
dim in, out as byte
out = ser.txbuffrq(10) ' Requesting 10 pages for the TX buffer. Out will
then contain how many can actually be allocated.

in = ser.rxbuffrq(7) ' Requesting 7 pages for the RX buffer. Will return
number of pages which can actually be allocated.

' .... Allocation requests for buffers of other objects ....

sys.buffalloc ' Performs actual memory allocation, as per previous
requests.
```

Actual memory allocation takes up to 100ms, so it is usually done just once, on boot, for all required buffers. If you do not require some buffer, you may choose not to allocate any memory to it. In effect, it will be disabled.

You may not always get the full amount of memory you have requested. Memory is not an infinite resource, and if you have already requested (and received) allocations for 95% of the memory for your platform, your next request will get up to 5% of memory, even if you requested for 10%.

There is a small overhead for each buffer. Meaning, not 100% of the memory allocated to a buffer is actually available for use. 16 bytes of each buffer are reserved for variables needed to administer this buffer, such as various pointers etc.

Thus, if we requested (and received) a buffer with 2 pages ($256 * 2 = 512$), we actually have 496 bytes in which to queue data ($512 - 16$).



You can only change the size of buffers that belong to serial ports that are closed ([ser.enabled](#)^[251] = 0) at the moment [sys.buffalloc](#) methods executes. If the port is opened at the time the [sys.buffalloc](#) method executes then buffer capacities for this port will remain unchanged, even if you requested changes through [ser.rxbuffrq](#) and [ser.txbuffrq](#).

Using Buffers

Once you have allocated memory for the TX and RX buffers you can start sending and receiving data through them.

Sending Data (through TX buffer)

Sending data is a two-step process. First, you put the data in the TX buffer using the [ser.setdata](#)^[264] method, and then you perform the actual sending (commit the data) using the [ser.send](#)^[264] method. For example:

```
ser.setdata ("Foo") ' Placed our data in the TX buffer - not being sent
out yet.

' ... more code...

ser.setdata ("Bar") ' Added even more data to our buffer, waiting to be
sent.

ser.send ' Now data will actually start going out. Data sent will be
'FooBar'.
```

Since this is a two-step process, you may first prepare a large block of data in the TX buffer and only then commit this data (this may come handy in some applications).



TiOS features *non-blocking operation*. This means that on ser.send, for example, the program does not halt and wait until the data is completely sent. In fact, execution resumes immediately, even before the first byte goes out. Your program will not freeze just because you ordered it to send a large chunk of data.

Receiving Data (through RX buffer)

Receiving data is a one-step process. To extract the data from a buffer, use the [ser.getdata](#)^[253] method. Data may only be extracted once from a buffer. Once extracted, it is no longer in the buffer. For example:

```
dim whatigot as string
whatigot = ser.getdata(255)
```

The variable whatigot now contains up to 255 bytes of data which came from the TX buffer of the serial port.

Buffer Memory Status

You cannot effectively use a buffer without knowing what its status is. Is it overflowing? Can you add more data? etc. Thus, each of the serial buffers has certain properties which allow you to monitor it:

The RX buffer

You can check the total capacity of the buffer with the [ser.rxbuffsize](#)^[262] property.

You can also find out how much *committed* data the RX buffer currently contains with the [ser.rxlen](#)^[263] property (see [Serial Settings](#)^[236] for explanation of what committed data is).

Sometimes you need to clear the RX buffer without actually extracting the data. In such cases the [ser.rxclear](#)^[263] comes in handy.

The TX buffer

Similarly to the RX buffer, the TX buffer also has a [ser.txbuffsize](#)^[264] property which lets you discover its capacity.

Unlike the RX buffer, the TX buffer has two "data length" properties: [ser.txlen](#)^[265] and [ser.newtxlen](#)^[256]. The *txlen* property returns the amount of *committed* data waiting to be sent from the buffer (you commit the data by using the [ser.send](#)^[264] method). The *newtxlen* property returns the amount of data which has entered the buffer, but has not yet been committed for sending.

The TX buffer also has a [ser.txfree](#)^[266] property, which directly tells you how much space is left in it. This does not take into account uncommitted data in the buffer -- actual free space is *ser.txfree*-*ser.newtxlen*!



$$\text{ser.txlen} + \text{ser.txfree} = \text{ser.txbuffsize}.$$

When you want to clear the TX buffer without sending anything, use the [ser.rxclear](#)^[266] method.

An example illustrating the difference between *ser.txlen* and *ser.newtxlen*:

```
sub on_sys_init
    dim x,y as word ' declare variables
    ser.rxbuffrq(1) ' Request one page for the rx buffer.
    ser.txbuffrq(5) ' Request 5 pages for the tx buffer (which we will use).
    sys.buffalloc ' Actually allocate the buffers.

    ser.setdata("foofoo") ' Set some data to send.
    ser.setdata("bar") ' Some more data to send.
    ser.send ' Start sending the data (commit).
    ser.setdata("baz") ' Some more data to send.
    x = ser.txlen ' Check total amount of data in the tx buffer.
    y = ser.newtxlen ' Check length of data not yet committed. Should be 3.

end sub 'Set up a breakpoint HERE.
```

Don't step through the code. The sending is fast -- by the time you reach *x* and *y* by stepping one line at a time, the buffer will be empty and *x* and *y* will be 0. Set a breakpoint at the end of the code, and then check the values for the variables (by using the [watch](#)^[33]).

Receiving Data

In a typical system, there is a constant need to handle an inflow of data. A simple approach is to use polling. You just poll the buffer in a loop and see if it contains any data, and when fresh data is available, you do something with it. This would look like this:

```

sub on_sys_init

while ser.rxlen = 0
wend ' basically keeps executing again and again as long as ser.rxlen = 0
s = ser.getdata(255) ' once loop is exited, it means data has arrived. We
extract it.

end sub

```

This approach will work, but it will forever keep you in a specific event handler (such as [on_sys_init](#)^[220]) and other events will never get a chance to execute. This is an example of *blocking code* which could cause a system to freeze. Of course, you can use the [doevents](#)^[82] statement, but generally we recommend you to avoid this approach.

Since our platform is event-driven, you should use events to tell you when new data is available. There is an [on_ser_data_arrival](#)^[258] event which is generated whenever there is data in the rx buffer:

```

sub on_ser_data_arrival

dim s as string
s = ser.getdata(255) ' Extract the data -- but in a non-blocking way.
' .... code to process data ....
end sub

```

This [on_ser_data_arrival](#)^[258] event is generated whenever there is data in the RX buffer, but only once. There are never two on_ser_data_arrival events waiting in the queue. The next event is only generated after the previous one has completed processing, if and when there is any data available in the RX buffer.

This means that when handling this event, you don't have to get *all* the data in the RX buffer. You can simply handle a chunk of data and once you leave the event handler, a new event of the same type will be generated if there is still unprocessed data left.

Here is a correct example of handling arriving serial data through the on_ser_data_arrival event. This example implements a data loopback -- whatever is received by the serial port is immediately sent back out.

```

sub on_ser_data_arrival
    ser.setdata(ser.getdata(ser.txfree))
    ser.send
end sub

```

We want to handle this loopback as efficiently as possible, but we must not overrun the TX buffer. Therefore, we cannot simply copy all arriving data from the RX buffer into the TX buffer. We need to check how much free space is available in the TX buffer. The first line of this code implements just that: [Ser.getdata](#)^[253] method takes as much data from the RX buffer as possible, but not more than [ser.txfree](#)^[266] (the available room in the TX buffer). The second line just sends the data.



Actually, this call will handle no more than 255 bytes in one pass. Even though we seemingly copy the data directly from the RX buffer to the TX buffer, this is done via a temporary string variable automatically created for this purpose. In this platform, string variables cannot exceed 255 bytes.



Polling method of data processing can sometimes be useful. See [Generating Dynamic HTML Pages](#)^[320].

Sending Data

In the previous section, we explained how to handle an incoming stream of data. You could say it was incoming-data driven. Sometimes you need just the opposite -- you need to perform operations based on the sending of data.

For example, supposing that in a certain system, you need to send out a long string of data when a button is pressed. A simple code for this would look like this:

```
sub on_button_pressed
    ser.setdata("This is a long string waiting to be sent. Send me
already!")
    ser.send
end sub
```

The code above *would* work, but *only* if at the moment of code execution the necessary amount of free space was available in the TX buffer (otherwise the data would get truncated). So, obviously, you need to make sure that the TX buffer has the necessary amount of free space before sending. A simple polling solution would look like this:

```
sub on_button_pressed
    dim s as string
    s = "This is a long string waiting to be sent. Send me already!"
    while ser.txfree < len(s) 'we will wait for the necessary amount of
free space to become available
    wend
    ser.setdata(s)
    ser.send
end sub
```

Again, this is not so good, as it would block other event handlers. So, instead of doing that, we would employ a code that uses [on_ser_data_sent](#)^[258]:

```

dim s as string
s = "This is a long string waiting to be sent. Send me already!"

sub on_button_pressed
    ser.notifysent(ser.txbuffsize-len(s)) ' causes the on_ser_data_sent
event to fire when the tx buffer has space for our string
end sub

sub on_ser_data_sent
    ser.setdata(s) ' put data in tx buffer
    ser.send ' start sending it.
end sub

```

When we press the button, [on_button_pressed](#)^[273] event is generated, so now the system knows we have a string to send. Using [ser.notifysent](#)^[254] we make the system fire the [on_ser_data_sent](#)^[258] event when the necessary amount of free space becomes available. This event will only be fired once -- and will be fired immediately if there is already enough available space.

Within the [on_ser_data_sent](#) event handler we put the data in the TX buffer and start sending it.



Amount of data that will trigger [on_ser_data_sent](#) does not include uncommitted data in the TX buffer.

Handling Buffer Overruns

Handling RX buffer overruns

The [on_ser_overrun_event](#)^[255] is generated when an RX buffer overrun has occurred. It means the data has been arriving to the UART faster than you were handling it and that some data got lost.

This event is generated just once, no matter how much data is lost. A new event will be generated only after exiting the handler for the previous one. In the UART/full-duplex mode (see [UART Mode](#)^[226] for details) you can typically prevent this from happening by using the flow control ([ser.flowcontrol](#)^[253]).

Typically, the user of your system wants to know when an overrun has occurred. For example, you could blink a red LED when this happens.

```

sub on_ser_overrun
    pat.play("R-R-R-R")
end sub

```

Are TX buffer overruns possible?

TX buffer overruns are not possible. The serial port won't let you overload its TX buffer. If you try to add more data to the TX buffer than the free space in the buffer allows to store then the data you are adding will be truncated.

See [Sending Data](#)^[243] for explanation on how to TX data correctly.

Redirecting Buffers

The following example appeared under [Receiving Data](#)^[24]:

```
sub on_ser_data_arrival
    ser.setdata(ser.getdata(ser.txfree))
    ser.send
end sub
```

This example shows how to send all data incoming to the RX buffer out from the TX buffer, in just two lines of code. However fast, this technique still passes all data through your BASIC code, even though you are not processing (altering, sampling) it in any way.

A much more efficient and advanced way to do this would be using a technique called *buffer redirection* (buffer shorting). With buffer shorting, instead of receiving the data into the RX buffer of your serial port, you are receiving it directly into the TX buffer of another object which is supposed to send out this data. This can be a serial object (the same port or a different one), a socket object, etc.

To use buffer shorting, you invoke the [ser.redir](#)^[26] method and specify the buffer to which the data is to be redirected. Once this is done, the `on_ser_data_arrival` event won't be generated at all, because the data will be going directly to the TX buffer that you have specified. As soon as the data enters this buffer, it will be automatically committed for sending.

The `ser.redir` method will only work if the serial port is closed ([ser.enabled](#)^[25]=0-NO) at the time when this method is executed. Therefore, it makes sense to check the result of `ser.enabled` execution, as in the example below:

```
sub on_sys_init
    if ser.redir(PL_REDIR_SER)= PL_REDIR_SER then
        'redirection succeeded
    else
        'redirection failed (perhaps, the port is opened?)
    end if
end sub
```

The performance advantage of buffer shorting is enormous, due to two factors: first, you are not handling the data programmatically, so the VM isn't involved at all. And second, the data being received is received directly into the TX buffer from which it is transmitted, so there is less copying in memory.

Of course you cannot do anything at all with this data -- you are just pumping it through. However, very often this is precisely what is needed! Additionally, you can still catch [escape sequences](#)^[25].

To stop redirection, you can use `ser.redir(0)`, which means "receive data into the RX buffer in a normal fashion".

Sinking Data

Sometimes it is desirable to ignore all incoming data while still maintaining the serial port opened. The [ser.sinkdata](#)^[26] property allows you to do just that.

Set the `ser.sinkdata` to 1- YES, and all incoming data will be automatically discarded. This means that the [on_ser_data_arrival](#)^[25] event will not be generated, reading [ser.rxlens](#)^[26] will always be returning zero, and so on. No data will be

reaching its destination even in case of [buffer redirection](#)^[245]. [Escape characters](#)^[236], however, will still be detected in the incoming data stream.

Properties, Methods, Events

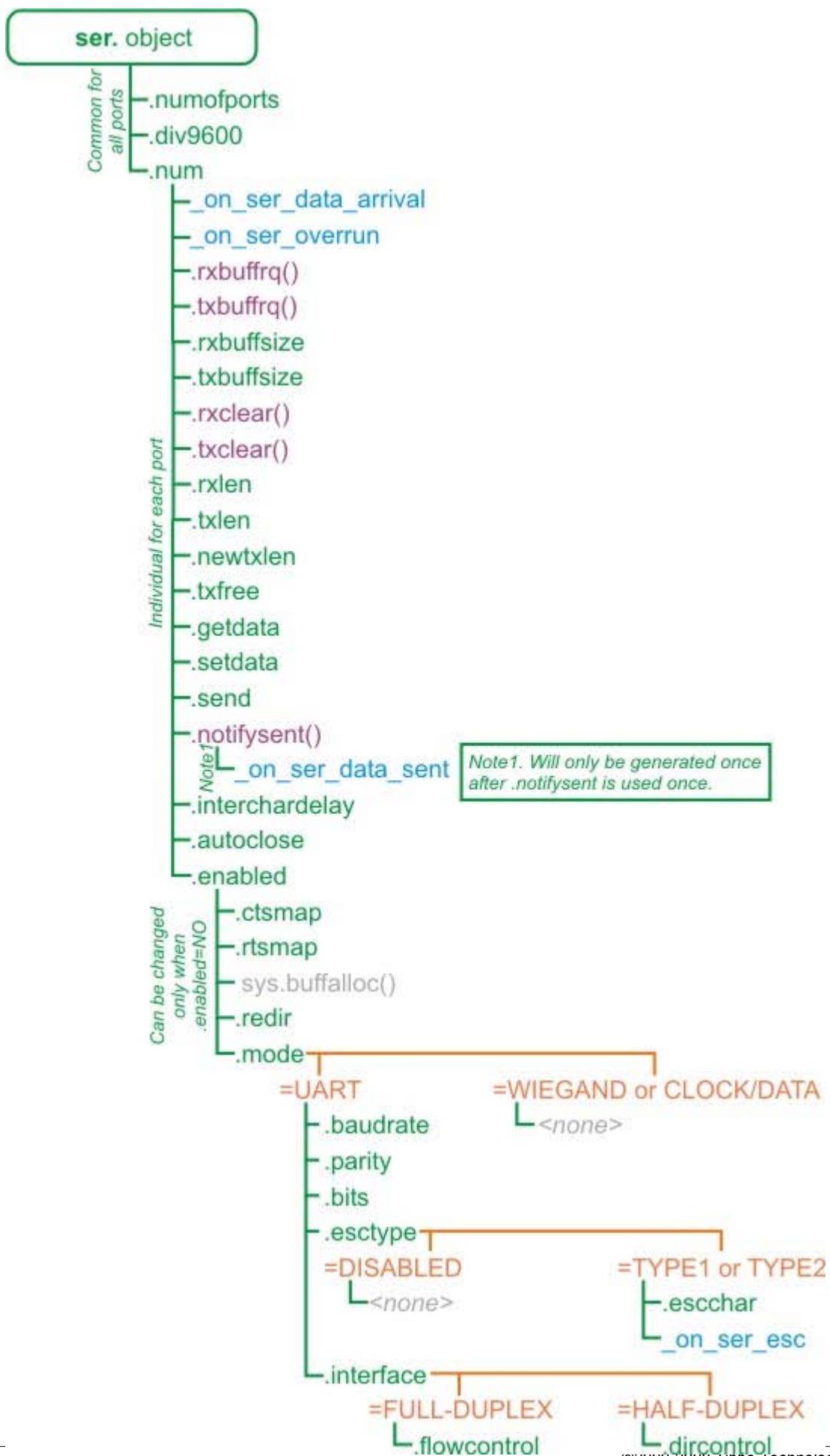
This section provides an alphabetical list of all properties, methods, and events of the ser object. For your convenience, here is a hierarchical map of the serial port's events, properties and methods.

All properties, methods, and events under [ser.num](#)^[257] are shown as sub-nodes of this property because they refer to a serial port currently selected by the ser.num. The [_on_ser_data_sent](#)^[258] event is subordinate to the [set.notifysent](#)^[257] method because this method needs to be called each time you want to receive the _on_ser_data_sent.

The [ser.ctsmap](#)^[249], [ser.rtsmap](#)^[261], [sys.buffalloc](#)^[217], [ser.redir](#)^[260], and [ser.mode](#)^[255] are subordinate to ser.enabled because they can be changed (or have effect) only when ser.enabled= 0- NO (the sys.buffalloc method is not a part of the ser object but its use is required for normal serial port operation- this is why it is listed here).

[Ser.baudrate](#)^[248], [ser.parity](#)^[260], [ser.bits](#)^[249], [ser.esctype](#)^[251], and [ser.interface](#)^[255] are only relevant in the UART mode of operation.

[Ser.escchar](#)^[251] is only relevant when ser.esctype is not DISABLED.



.Autoclose Property

Function:	For currently selected serial port (selection is made through ser.num) specifies whether the port will be disabled once the intercharacter gap expires.
Type:	dis_en (enum, byte)
Value Range:	0- DISABLED (default) 1- ENABLED
See Also:	Serial Settings

Details

The serial port is disabled by setting [ser.enabled](#)= 0- NO. Intercharacter gap duration is specified by the [ser.interchardelay](#) property.

This property offers a way to make sure that no further data is received once the gap of certain length is encountered. This property is especially useful in [Wiegand](#) or [clock/data](#) mode ([ser.mode](#)= 1- PL_SER_MODE_WIEGAND or 2- PL_SER_MODE_CLOCKDATA) where intercharacter gap is the only way to reliably identify the end of one data transmission.

.Baudrate Property

Function:	Sets/returns the baudrate "divisor value" for the selected serial port (selection is made through ser.num).
Type:	Word
Value Range:	0-65535, default value is platform dependent, results in 9600bps .
See Also:	UART Mode , Serial Settings

Details

Actual baudrate is calculated as follows: $(9600 * \text{ser.div9600}) / \text{ser.baudrate}$. The [ser.div9600](#) read-only property returns the value [ser.baudrate](#) must be set to in order to obtain 9600 bps on a particular platform.

For example, if we need to achieve 38400bps and [ser.div9600](#) returns 12, then we need to set [ser.baudrate](#)=3, because $(9600 * 12) / 38400 = 3$. This serves as a platform-independent baudrate calculation, as [ser.div9600](#) will return different values for different platforms.

This property is only relevant when the serial port is in the UART mode ([ser.mode](#)= 0- PL_SER_MODE_UART).



Technically speaking, this property should be called divisor, not baudrate. We called it *baudrate* so that you could easily find it.

.Bits Property

Function: Specifies the number of data bits in a word TXed/RXed by the serial port for the currently selected port (selection is made through [ser.num](#)^[257])

Type: Enum (pl_ser_bits, byte)

Value Range: 0- PL_SER_BB_7: data word TXed/RXed by the serial port is to contain 7 data bits

1- PL_SER_BB_8 (**default**): data word TXed/RXed by the serial port is to contain 8 data bits.

See Also: [UART Mode](#)^[226], [Serial Settings](#)^[236]

Details

This property is only relevant when the serial port is in the UART mode ([ser.mode](#)^[255]= 0- PL_SER_MODE_UART).

.Ctsmap property (Selected Platforms Only)

Function: Sets/returns the number of the I/O line that will act as **CTS/W0&1in/cin** input of currently selected serial port (selection is made through [ser.num](#)^[257]).

Type: Enum (pl_int_num, byte)

Value Range: Platform-specific, see the list of pl_int_num constants in the platform specifications.

See Also: [Three modes of the Serial Port](#)^[225], [Serial Settings](#)^[236]

Details



This property is only available on selected platforms. See "Platform-dependent Programming Information" topic inside your platform specifications section.

Default value of this property is different for each serial port. See the list of pl_int_num constants in the platform specifications -- it shows default values as well.

Selection can be made only among interrupt lines. Regular, non-interrupt I/O lines cannot be selected. Property value can only be changed when the port is closed ([ser.enabled](#)^[254]=0- NO).

On certain platforms, you may need to configure the line as input. This is done through the [io.enabled](#)^[370] property of the [io](#)^[365] object. See "Platform-dependent

"Programming Information" topic inside your platform specifications section.

.Dircontrol Property

Function:	Sets/returns the polarity of the direction control line (RTS) for selected serial port (selection is made through ser.num ^[257]).
Type:	Enum (pl_ser_dircontrol, byte)
Value Range:	0- PL_SER_DCP_LOWFORINPUT (default): The RTS output will be LOW when the serial port is ready to RX data and HIGH when the serial port is TXing data. 1- PL_SER_SI_HIGHFORINPUT: The RTS output will be HIGH when the serial port is ready to rx data and LOW when the serial port is txing.
See Also:	UART Mode ^[226] , Serial Settings ^[236]

Details

This property is only relevant in the UART/half-duplex mode ([ser.mode](#)^[255]= 0-PL_SER_MODE_UART and [ser.interface](#)^[255]= 1-PL_SER_SI_HALFDUPLEX).

Note, that HIGH/LOW states specified above are for the TTL-serial interface of the MODULE-level products. If you are dealing with the RS232 port then the states will be in reverse (for example, 1- PL_SER_SI_HIGHFORINPUT will mean "RTS LOW for input, HIGH for output").

Depending on your platform, you may be allowed to remap RTS line to other I/O pins of the device through the [ser.rtsmap](#)^[261] and [ser.ctsmap](#)^[249] properties. Also, you may be required to correctly configure RTS line as an input through the [io.enabled](#)^[370] property of the [io](#)^[365] object. See "Platform-dependent Programming Information" topic inside your platform specifications section.

When the serial port is in the UART/half-duplex mode you can use the CTS line as a regular I/O line of your device.

.Div9600 R/O Property

Function:	Returns the value to which the ser.baudrate ^[248] property must be set in order to achieve the baudrate of 9600bps on the current device and under present conditions.
Type:	Word
Value Range:	---
See Also:	Serial Settings ^[236]

Details

"Smart" applications will use this property to set baudrates in a platform-independent fashion. Even for the same device, the value required to achieve

9600bps may be different at different times. For example, some devices have PLLs (see [sys.currentpll](#)^[218]). Enabling and disabling PLL changes the clock frequency of the device and this affects the value returned by ser.div9600.

.Enabled Property

Function:	Enables/disables currently selected serial port (selection is made through ser.num ^[257]).
Type:	Enum (no_yes, byte)
Value Range:	0- NO (default): not enabled 1- YES: enabled
See Also:	Buffer Memory Status ^[240]

Details

Enabling/disabling the serial port does not automatically clear its buffers, this is done via [ser.rxclear](#)^[263] and [ser.txclear](#)^[266]. Notice that certain properties can only be changed and methods executed when the port is not enabled. These are [ser.rtsmap](#)^[261], [ser.ctsmap](#)^[249], [ser.mode](#)^[255], [ser.redirect](#)^[260]. You also cannot allocate buffer memory for the port (do [sys.buffalloc](#)^[217]) when the port is enabled.

.Escchar Property

Function:	For selected serial port (selection is made through ser.num ^[257]) sets/retrieves ASCII code of the escape character used for type1 or type2 serial escape sequences.
Type:	Byte
Value Range:	0-255, default=1 (SOH character)
See Also:	UART Mode ^[226] , Serial Settings ^[236]

Details

Which escape sequence is enabled is defined by the [ser.esctype](#)^[251] property. This property is irrelevant when ser.esctype= 0- PL_SER_ET_DISABLED or when the serial port is in the [Wiegand](#)^[229] or [clock/data](#)^[232] mode (ser.mode= 1- PL_SER_MODE_WIEGAND or ser.mode= 2- PL_SER_MODE_CLOCKDATA) -- serial escape sequences are only recognized in the UART data.

.Esctype Property

Function:	Defines, for selected serial port (selection is made through ser.num ^[257]), whether serial escape sequence recognition is enabled and, if yes, what type of escape sequence is to be recognised.
------------------	---

Type:	Enum (pl_ser_esctype, byte)
Value Range:	0- PL_SER_ET_DISABLED (disabled): Recognition of serial escape sequences disabled. 1- PL_SER_ET_TYPE1: Escape sequences of type 1 are to be recognized. 2- PL_SER_ET_TYPE2: Escape sequences of type 2 are to be recognized.
See Also:	UART Mode ^[226] , Serial Settings ^[236]

Details

Escape sequence is a special occurrence of characters in the incoming data received by the serial port. Escape sequences are only recognized in the [UART](#)^[226] mode of operation ([ser.mode](#)^[255]= 0- PL_SER_MODE_UART).

When escape sequence is detected the [on_ser_esc](#)^[259] event is generated and the serial port is disabled ([ser.enabled](#)^[251]= 0- NO). When enabled, serial escape sequence detection works even when the [buffer shorting](#)^[245] is employed (see [ser.redir](#)^[260] property).

Follows is the description of two escape sequence types:

Type 1 Type1 escape sequence consists of three consecutive escape characters (ASCII code of escape character is defined by the [ser.escchar](#)^[251] property). For the escape sequence to be recognized each of the escape characters must be preceded by a time gap of at least 100ms:

...previous <--100ms-- E.C. <--100ms-- E.C. <--100ms-- E.C.
 data > > >

If the time gap before a certain escape character exceeds 100ms then this character is considered to be a part of the escape sequence and is not recorded into the RX buffer. If the time gap before a certain escape character is less than 100ms than this character is considered to be a normal data character and is saved into the RX buffer. Additionally, escape character counter is reset and the escape sequence must be started again. The following example illustrates one important point (escape characters are shown as ■). Supposing the serial port receives the following string:

ABC<--100ms-->■<--100ms-->■■DE

First two escape characters in this example had correct time gap before them, so they were counted as a part of the escape sequence and not saved into the buffer. The third escape character did not have a correct time gap so it was interpreted as a data character and saved into the buffer. The following was recorded into the RX buffer:

ABC■DE

The side effect and the point this example illustrates is that first two escape characters were lost -- they neither became a part of a successful escape sequence (because this sequence wasn't completed), nor were saved into the buffer.

Type 2 Type 2 escape sequence is not based on any timing. Escape sequence consists of escape character (defined by the `ser.escchar` property) followed by any character other than escape character. To receive a *data* character whose ASCII code matches that of escape character the serial port must get this character *twice*. This will result in a single character being recorded into the RX buffer.

The following sequence will be recognized as escape sequence (that is, if current escape character is not 'D'):

ABC ■ D

In the sequence below two consecutive escape characters will be interpreted as data (data recorded to the RX buffer will contain only one such character):

ABC ■ ■

.Flowcontrol Property

Function:	Sets/returns flow control mode for currently selected serial port (selection is made through ser.num)
Type:	<code>dis_en</code> (enum, byte)
Value Range:	0- DISABLED (default) 1- ENABLED
See Also:	UART Mode , Serial Settings

Details

Only relevant when the `ser.mode`= 0- PL_SER_MODE_UART and `ser.interface`= 0- PL_SER_SI_FULLDUPLEX (full-duplex). Flow control uses two serial port lines- RTS and CTS- to regulate the flow of data between the serial port of your device and another ("attached") serial device.

Depending on your platform, you may be allowed to remap RTS and CTS lines to other I/O pins of the device through the `ser.rtsmap` and `ser.ctsmap` properties. Also, you may be required to correctly configure RTS and CTS lines as an input and output through the `io.enabled` property of the `io` object. See "Platform-dependent Programming Information" topic inside your platform specifications section.

When the flow control is disabled both RTS and CTS lines are not used by the serial port and become regular I/O lines that can be controlled through the `io` object.

.Getdata Method

Function:	For the selected serial port (selection is made through ser.num) returns the string that contains the data extracted from the RX buffer.
Syntax:	ser.getdata(maxinplen as word) as string
Returns:	String containing data extracted from the RX buffer

See Also:[Three Modes of the Serial Port](#)^[225], [Receiving data](#)^[241]

Part	Description
maxinplen	Maximum amount of data to return (word).

Details

Extracted data is permanently deleted from the buffer. Length of extracted data is limited by one of the three factors (whichever is smaller): amount of committed data in the RX buffer itself, capacity of the "receiving" string variable, and the limit set by the maxinplen argument.

In the [UART](#)^[226] mode ([ser.mode](#)^[255]= 0- PL_SER_MODE_UART) the data is extracted "as is". For [Wiegand](#)^[229] and [clock/data](#)^[232] mode (ser.mode= 1- PL_SER_MODE_WIEGAND and ser.mode= 2- PL_SER_MODE_CLOCKDATA) each character of extracted data represents one data bit and only two characters are possible: '0' or '1'.

.Interchardelay Property

Function:	Sets/returns maximum intercharacter delay for the selected serial port (selection is made through ser.num ^[257]) in 10ms steps.
Type:	Byte
Value Range:	0-255, default = 0 (no delay)
See Also:	Three Modes of the Serial Port ^[225] , Serial Settings ^[236]

Details

For UART mode ([ser.mode](#)^[255]= 0- PL_SER_MODE_UART) specifies the time that needs to elapse since the arrival of the most recent serial character into the RX buffer to cause the data to be committed (and [on_ser_data_arrival](#)^[258] event generated). For Wiegand and clock/data mode (ser.mode= 1- PL_SER_MODE_WIEGAND or 2- PL_SER_MODE_CLOCKDATA) the time since the most recent data bit (high-to-low transition on the **W0&1in/cin** line) is counted.

In the UART mode this property allows you to combine incoming serial data into larger "chunks", which typically improves performance. Notice, that the intercharacter gap is not counted when the new data is not being received because the serial port has set the RTS line to LOW (not ready). For this to happen, the serial port must be in the UART/full-duplex/flow control mode (ser.mode= 0- PL_SER_MODE_UART, [ser.interface](#)^[255]= 0- 0- PL_SER_SI_FULLDUPLEX, and [ser.flowcontrol](#)^[253]= 1- ENABLED) and the RX buffer must be getting nearly full (less than 64 bytes of free space left).

For Wiegand and clock/data modes, counting timeout since the last bit is the only way to determine the end of the data output. Suggested timeout is app. 10 times the bit period of the data output by attached Wiegand or clock/data device.

.Interface Property

Function:	Chooses full-duplex or half-duplex operating mode for currently selected serial port (selection is made through ser.num ^[257]).
Type:	Enum (pl_ser_interface, byte)
Value Range:	0- PL_SER_SI_FULLDUPLEX (default): full-duplex mode. 1- PL_SER_SI_HALFDUPLEX: half-duplex mode.
See Also:	UART Mode ^[226] , Serial Settings ^[236]

Details

Full-duplex mode is suitable for RS232, RS422, or four-wire RS485 communications. RTS output (together with CTS input) can be used for optional hardware flow control ([ser.flowcontrol](#)^[253]).

Half-duplex mode is suitable for two-wire RS485 communications. RTS line is used for direction control. Hardware flow control is not possible, so ser.flowcontrol value is irrelevant. Direction control polarity can be set through [ser.dircontrol](#)^[250].

This property is only relevant when the port is in the UART mode ([ser.mode](#)^[255]= 0-PL_SER_MODE_UART).

Depending on your platform, you may be allowed to remap RTS and CTS lines to other I/O pins of the device through the [ser.rtsmap](#)^[261] and [ser.ctsmap](#)^[249] properties. Also, you may be required to correctly configure RTS and CTS lines as an input and output through the [io.enabled](#)^[370] property of the [io](#)^[365] object. See "Platform-dependent Programming Information" topic inside your platform specifications section.

.Mode Property

Function:	Sets operating mode for the currently selected serial port (selection is made through ser.num ^[257]).
Type:	Enum (pl_ser_mode, byte)
Value Range:	0- PL_SER_MODE_UART (default): UART mode. 1- PL_SER_MODE_WIEGAND: Wiegand mode. 2- PL_SER_MODE_CLOCKDATA: clock/data (magstripe interface) mode.
See Also:	Three Modes of the Serial Port ^[225] , Serial Settings ^[236]

Details

Follows is a short introduction of three operating modes of the serial port:

UART mode	Suitable for RS232, RS422, RS485, etc. communications in full-duplex or half-duplex mode (see ser.interface ^[255]). Data is
------------------	---

transmitted through the **TX** pin and received through the **RX** pin. Optionally, **RTS** (output) and **CTS** (input) lines are used for flow control (see [ser.flowcontrol](#)^[253]) in the full-duplex mode. Additionally, RTS can be used for direction control in the half-duplex mode.

Wiegand mode	Suitable for sending to or receiving data from any standard Wiegand device. Data transmission is through pins W0out and W1out , reception- through W0&1in and W1in . "W0&1in" means that a logical AND of W0 and W1 signals must be applied to this input. Therefore, external logical gate is needed in order to receive Wiegand data.
Clock/data mode	Suitable for sending to or receiving data from any standard clock/data (or magstripe) device. Data transmission is through pins cout and dout , reception- through cin and din . Third line of the magstripe interface- card present- is not required for data reception. For transmission, any I/O line can be used as card present output (under software control).

Changing port mode is only possible when the port is closed ([ser.enabled](#)^[254]= 0-NO). Depending on your platform, you may be allowed to remap RTS/W1out/cout and CTS/W0&1in/cin lines to other I/O pins of the device through the [ser.rtsmap](#)^[261] and [ser.ctsmap](#)^[249] properties. Also, depending on selected mode and your platform you may be required to correctly configure some of your serial port's lines as inputs or outputs through the [io.enabled](#)^[370] property of the [io](#)^[365] object. See "Platform-dependent Programming Information" topic inside your platform specifications section.



We understand that it would be much more logical to call this property "interface", not "mode". Problem is, this property was added later, when `ser.interface` already came to mean something [else](#)^[255]. So, we had no choice but to choose unused term.

.Newtxlen R/O Property

Function:	For the selected serial port (selection is made through ser.num ^[257]) returns the amount of uncommitted TX data in bytes.
Type:	Word
Value Range:	0-65535, default = 0 (bytes)
See Also:	Sending (Transmitting) Data ^[243] , ser.txlen ^[267] , ser.txfree ^[266]

Details

Uncommitted data is the one that was added to the TX buffer with the [ser.setdata](#)^[264] method but not yet committed using the [ser.send](#)^[264] method.

.NotifySent Method

Function:	Using this method for the selected serial port (selection is made through ser.num ^[257]) will cause the on_ser_data_sent ^[258] event to be generated when the amount of committed data in the TX buffer is found to be below "threshold" number of bytes.
Syntax:	notifysent(threshold as word)
Returns:	---
See Also:	Sending (Transmitting) Data ^[243]

Part	Description
threshold	Amount of bytes in the TX buffer below which the event it so be generated.

Details

Only one `on_ser_data_sent` event will be generated each time after the `ser.notifysent` method is invoked. This method, together with the `on_ser_data_sent` event provides a way to handle data sending asynchronously. Just like with [ser.txfree](#)^[266], the trigger you set won't take into account any uncommitted data in the TX buffer.

.Num Property

Function:	Sets/returns the number of currently selected serial port (ports are enumerated from 0).
Type:	Byte
Value Range:	The value of this property won't exceed ser.numofports ^[258] -1 (even if you attempt to set higher value). Default = 0 (port #0 selected)
See Also:	---

Details

All other properties and methods of this object relate to the serial port selected through this property. Note that serial port-related events such as [on_ser_data_arrival](#)^[258] change currently selected port!

.Numofports R/O Property

Function:	Returns total number of serial ports found on the current platform.
Type:	Byte
Value Range:	platform-dependent
See Also:	ser.num <small>[257]</small>

Details

On_ser_data_arrival Event

Function:	Generated when at least one data byte is present in the RX buffer of the serial port (i.e. for this port the ser.rxlen <small>[263]</small> > 0).
Declaration:	on_ser_data_arrival
See Also:	Buffer Memory Status <small>[240]</small>

Details

When the event handler for this event is entered the [ser.num](#) [257] property is automatically switched to the port for which this event was generated. Another [on_ser_data_arrival](#) [258] event on a particular port is never generated until the previous one is processed. Use [ser.getdata](#) [253] method to extract the data from the RX buffer.

You don't have to process all data in the RX buffer at once. If you exit the [on_ser_data_arrival](#) event handler while there is still some unprocessed data in the RX buffer another [on_ser_data_arrival](#) event will be generated immediately.

This event is not generated for a particular port when buffer redirection is set for this port through the [ser.redir](#) [260] method.

On_ser_data_sent Event

Function:	Generated after the total amount of <i>committed</i> data in the TX buffer of the serial port (ser.txlen <small>[267]</small>) is found to be less than the threshold that was preset through the ser.notifysent <small>[257]</small> method.
Declaration:	on_ser_data_sent
See Also:	Sending (Transmitting) Data <small>[243]</small>

Details

This event may be generated only after the ser.notifysent method was used. Your application needs to use the ser.notifysent method EACH TIME it wants to cause the on_ser_data_sent event generation for a particular port. When the event handler for this event is entered the [ser.num](#)²⁵⁷ is automatically switched to the port on which this event was generated.

Please, remember that uncommitted data in the TX buffer is not taken into account for on_ser_data_sent event generation.

On_ser_esc Event

Function:	Generated when currently enabled escape sequence is detected in the RX data stream.
Declaration:	on_ser_esc
See Also:	Serial Settings ²³⁶

Details

Once the serial escape sequence is detected on a certain serial port this port is automatically disabled (ser.enabled= 0- NO).

When event handler for this event is entered the [ser.num](#)²⁵⁷ property is automatically switched to the port on which this event was generated.

Whether or not escape sequence detection is enabled and what kind of escape sequence is expected is defined by the [ser.esctype](#)²⁵⁸ property. Escape sequence detection works even when buffer redirection is set for the serial port using the [ser.redir](#)²⁶⁰ method.

On_ser_overrun Event

Function:	Generated when data overrun has occurred in the RX buffer of the serial port.
Declaration:	on_ser_overrun
See Also:	Handling Buffer Overruns ²⁴⁴

Details

Another on_ser_overrun event for a particular port is never generated until the previous one is processed. When the event handler for this event is entered the [ser.num](#)²⁵⁷ property is automatically switched to the port on which this event was generated.

Data overruns are a common occurrence on serial lines. The overrun happens when the serial data is arriving into the RX buffer faster than your application is able to extract it, the buffer runs out of space and "misses" some incoming data.

Data overruns are typically prevented through the use of RTS/CTS flow control (see the [ser.flowcontrol](#)²⁵³ property).

.Parity Property

Function:	Sets/returns parity mode for the selected serial port (selection is made through ser.num ²⁵⁷)
Type:	Enum (pl_ser_parity, byte)
Value Range:	0- PL_SER_PR_NONE: no parity bit to be transmitted. 1- PL_SER_PR_EVEN: even parity. 2- PL_SER_PR_ODD: odd parity. 3- PL_SER_PR_MARK: parity bit always at "1". 4- PL_SER_PR_SPACE: parity bit always at "0".

See Also: [UART Mode](#)²²⁶, [Serial Settings](#)²³⁶

Details

Mark parity is equivalent to having a second stop-bit (there is no separate property to explicitly select the number of stop bits).

This property is only relevant when the serial port is in the UART mode ([ser.mode](#)²⁵⁵= 0- PL_SER_MODE_UART).

.Redir Method

Function:	For the selected serial port (selection is made through ser.num ²⁵⁷) redirects the data being RXed to the TX buffer of the same serial port, different serial port, or another object that supports compatible buffers.
Syntax:	ser.redir(redir as pl_redir) as pl_redir
Returns:	Returns 0- PL_REDIR_NONE if redirection failed or the same value as was passed in the redir argument.
See Also:	Redirecting Buffers (Shorting) ²⁴⁵

Part	Description
redir	Platform-specific. See the list of pl_redir constants in the platform specifications.

Details

Data redirection (sometimes referred to as "buffer shorting") allows to arrange efficient data exchange between ports, sockets, etc. in cases where no data alteration or parsing is necessary, while achieving maximum possible throughput is important.

The redir argument, as well as the value returned by this method are of "enum pl_redir" type. The pl_redir defines a set of platform inter-object constants that

include all possible redirections for this platform. Specifying redir value of 0-PL_REDIR_NONE cancels redirection. When the redirection is enabled for a particular serial port, the [on_ser_data_arrival](#)^[258] event is not generated for this port.

Once the RX buffer is redirected certain properties and methods related to the RX buffer will actually return the data for the TX buffer to which this RX buffer was redirected:

- [Ser.rxbuffsize](#)^[262] will actually be returning the size of the TX buffer.
- [Ser.rxclear](#)^[263] method will actually be clearing the TX buffer.
- [Ser.rxlen](#)^[263] method will be showing the amount of data in the TX buffer.



If the redirection is being done on a serial port that is currently opened ([ser.enabled](#)^[251]= 1- YES) then this port will be closed automatically.

.Rtsmap Property (Selected Platforms Only)

Function:	Sets/returns the number of the I/O line that will act as RTS/W0out/cout output of currently selected serial port (selection is made through ser.num ^[257]).
Type:	Enum (pl_io_num, byte)
Value Range:	Platform-specific, see the list of pl_io_num constants in the platform specifications.
See Also:	Three modes of the Serial Port ^[225] , Serial Settings ^[236]

Details



This property is only available on selected platforms. See "Platform-dependent Programming Information" topic inside your platform specifications section.

Default value of this property is different for each serial port. See the list of pl_int_num constants in the platform specifications -- it shows default values as well.

Absolutely any I/O line can be selected by this property, as long as this line is not occupied by some other function. Property value can only be changed when the port is closed ([ser.enabled](#)^[251]= 0- NO).

On certain platforms, you may need to configure the line as output. This is done through the [io.enabled](#)^[370] property of the [io](#)^[365] object. See "Platform-dependent Programming Information" topic inside your platform specifications section.

.Rxbuffrq Method

Function:	For the selected serial port (selection is made through ser.num ^[257]) pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the RX buffer of the serial port.
Syntax:	ser.rxbuffrq(numpages as byte) as byte
Returns:	Actual number of pages that can be allocated (byte)
See Also:	Allocating Memory for Buffers ^[239] , ser.txbuffrq ^[265]

Part	Description
numpages	Requested numbers of buffer pages to allocate.

Details

Returns actual number of pages that can be allocated. Actual allocation happens when the [sys.buffalloc](#)^[217] method is used. The serial port is unable to RX data if its RX buffer has 0 capacity. Actual current buffer capacity can be checked through the [ser.rxbuffsize](#)^[262] which returns buffer capacity in bytes.

Relationship between the two is as follows: ser.rxbuffsize=num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the ser.rxbuffrq. "-16" is because 16 bytes are needed for internal buffer variables.

Buffer allocation will not work if the serial port to which this buffer belongs is opened ([ser.enabled](#)^[251]= 1- YES) at the time when sys.buffalloc executes. You can only change buffer sizes of ports that are closed.

.Rxbuffsize R/O Property

Function:	For the selected serial port (selection is made through ser.num ^[257]) returns current RX buffer capacity in bytes.
Type:	Word
Value Range:	0-65535
See Also:	Buffer Memory Status ^[240]

Details

Buffer capacity can be changed through the [ser.rxbuffrq](#)^[262] method followed by the [sys.buffalloc](#)^[217] method.

The ser.rxbuffrq requests buffer size in 256-byte pages whereas this property returns buffer size in bytes. Relationship between the two is as follows: ser.rxbuffsize=num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the ser.rxbuffrq. "-16" is

because 16 bytes are needed for internal buffer variables. The serial port cannot RX data when the RX buffer has zero capacity.

.Rxclear Method

Function:	For the selected serial port (selection is made through ser.num ^[257]) clears (deletes all data from) the RX buffer.
Syntax:	ser.rxclear
Returns:	---
See Also:	Buffer Memory Status ^[240]

Details

.Rxlen R/O Property

Function:	For the selected serial port (selection is made through ser.num ^[257]) returns total number of committed bytes currently waiting in the RX buffer to be extracted and processed by your application.
Type:	Word
Value Range:	0-65535
See Also:	Serial Settings ^[236] , Buffer Memory Status ^[240]

Details

The [on_ser_data_arrival](#)^[258] event is generated once the RX buffer is not empty, i.e. there is data to process. There may be only one [on_ser_data_arrival](#) event for each port waiting to be processed in the event queue. Another [on_serial_data_arrival](#) event for the same port may be generated only after the previous one is handled.

If, during the [on_ser_data_arrival](#) event handler execution, not all data is extracted from the RX buffer, another [on_ser_data_arrival](#) event is generated immediately after the [on_ser_data_arrival](#) event handler is exited.

Notice that the RX buffer of the serial port employs "data committing" based on the amount of data in the buffer and intercharacter delay ([ser.interchardelay](#)^[254]). Data in the RX buffer may not be committed yet. Uncommitted data is not visible to your application and is not included in the count returned by the [ser.rxlen](#).

.Send Method

Function:	For the selected serial port (selection is made through ser.num) commits (allows sending) the data that was previously saved into the TX buffer using the ser.setdata method.
Syntax:	ser.send
Returns:	---
See Also:	Serial Settings

Details

You can monitor the sending progress by checking the [ser.txlen](#) property or using the [ser.notifysent](#) method and the [on_ser_data_sent](#) event.

.Setdata Method

Function:	For the selected serial port (selection is made through ser.num) adds the data passed in the txdata argument to the contents of the TX buffer.
Syntax:	ser.setdata(byref txdata as string)
Returns:	---
See Also:	Three Modes of the Serial Port , Sending Data , ser.txlen , ser.txfree ,

Part	Description
txdata	The data to send.

Details

In the [UART](#) mode ([ser.mode](#)= 0- PL_SER_MODE_UART) the data is added "as is". For [Wiegand](#) and [clock/data](#) mode ([ser.mode](#)= 1- PL_SER_MODE_WIEGAND and [ser.mode](#)= 2- PL_SER_MODE_CLOCKDATA) each data character represents one data bit and only bit0 (least significant bit) of each character is relevant (therefore, adding "0101" will result in the 0101 sequence of data bits).

If the buffer doesn't have enough space to accommodate the data being added then this data will be truncated. Newly saved data is not sent out immediately. This only happens after the [ser.send](#) method is used to commit the data. This allows your application to prepare large amounts of data before sending it out.

Total amount of newly added (uncommitted) data in the buffer can be checked through the [ser.newtxlen](#) setting.

.Sinkdata Property

Function:	For the currently selected serial port (selection is made through ser.num ^[257]) specifies whether the incoming data should be discarded.
Type:	Enum (yes_no, byte)
Value Range:	0- NO (default): normal data processing. 1- YES: discard incoming data.
See Also:	Sinking Data ^[245]

Details

Setting this property to 1- YES causes the serial port to automatically discard all incoming data without passing it to your application. The [on_ser_data_arrival](#)^[258] event will not be generated, reading [ser.rxlen](#)^[263] will always return zero, and so on. No data will be reaching its destination even in case of [buffer redirection](#)^[245]. [Escape characters](#)^[236], however, will still be detected in the incoming data stream.

.Txbuffrq Method

Function:	For the selected serial port (selection is made through ser.num ^[257]) pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the TX buffer of the serial port.
Syntax:	ser.txbuffrq(numpages as byte) as byte
Returns:	Actual number of pages that can be allocated (byte).
See Also:	Allocating Memory for Buffers ^[239] , ser.rxbuffrq ^[262]

Part	Description
numpages	Requested numbers of buffer pages to allocate.

Details

Returns actual number of pages that can be allocated. Actual allocation happens when the [sys.buffalloc](#)^[217] method is used. The serial port is unable to TX data if its TX buffer has 0 capacity. Actual current buffer capacity can be checked through the [ser.txbuffsize](#)^[266] which returns buffer capacity in bytes.

Relationship between the two is as follows: ser.txbuffsize=num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the ser.txbuffrq. "-16" is because 16 bytes are needed for internal buffer variables.

Buffer allocation will not work if the serial port to which this buffer belongs is opened ([ser.enabled](#)^[251]= 1- YES) at the time when sys.buffalloc executes. You can only change buffer sizes of ports that are closed.

.Txbuffsize R/O Property

Function:	For the selected serial port (selection is made through ser.num) returns current TX buffer capacity in bytes.
Type:	Word
Value Range:	0-65535
See Also:	Buffer Memory Status

Details

Buffer capacity can be changed through the [ser.txbuffrq](#) method followed by the [sys.buffalloc](#) method.

The ser.txbuffrq requests buffer size in 256-byte pages whereas this property returns buffer size in bytes. Relationship between the two is as follows: ser.txbuffsize=num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the ser.txbuffrq. "-16" is because 16 bytes are needed for internal buffer variables. The serial port cannot TX data when the TX buffer has zero capacity.

.Txclear Method

Function:	For the selected serial port (selection is made through ser.num) clears (deletes all data from) the TX buffer.
Syntax:	ser.txclear
Returns:	---
See Also:	Buffer Memory Status

Details

This method will **only** work when the port is closed ([Ser.Enabled Property](#) = NO). You cannot clear the TX buffers while the port is open.

.Txfree R/O Property

Function:	For the selected serial port (selection is made through ser.num) returns the amount of free space in the TX buffer in bytes.
Type:	Word
Value Range:	0-65535
See Also:	Buffer Memory Status

Details

Notice, that the amount of free space returned by this property does not take into account any uncommitted data that might reside in the buffer (this can be checked via [ser.newtxlen](#)^[256]). Therefore, actual free space in the buffer is ser.txfree-ser.newtxlen. Your application will not be able to store more data than this amount.

To achieve asynchronous data processing, use the [ser.notifysent](#)^[257] method to get [on_ser_data_sent](#)^[258] event once the TX buffer gains required amount of free space.

.Txlen R/O Property

Function:	For the selected serial port (selection is made through ser.num ^[257]) returns total number of <i>committed</i> bytes currently found in the TX buffer.
Type:	Word
Value Range:	0-65535
See Also:	Serial Settings ^[236] , Buffer Memory Status ^[240] , ser.newtxlen ^[256]

Details

The data in the TX buffer does not become committed until you use the [ser.send](#)^[264] method.

Your application may use the [ser.notifysent](#)^[257] method to get [on_ser_data_sent](#)^[258] event once the total number of committed bytes in the TX buffer drops below the level defined by the ser.notifysent method.

Net Object



The net object represents the Ethernet interface of your device. This object only specifies various parameters related to the Ethernet interface and is not responsible for sending/transmitting network data. The latter is the job of the [sock](#)^[274] object.

Here is what you can do with the net object:

- Check if the Network Interface Controller (NIC) IC is functioning properly.
- Check your device's Ethernet MAC address.
- Set the IP-address of the Ethernet Interface.
- Set default gateway IP and the netmask.
- Be notified when the Ethernet cable is plugged or unplugged and check current

link status.

- Be notified when data overflow occurs in the NIC.

Overview 3.3.1

Here you will find:

- [Main Parameters](#)^[268] (IP, gateway IP, netmask, MAC).
- [Checking Ethernet status](#)^[269] (link status change, failure, overflows).

Main Parameters

To enable Ethernet communications, you need to set the [net.ip](#)^[270], [net.gatewayip](#)^[270], and the [net.netmask](#)^[270] properties. Actually, `net.gatewayip` and `net.netmask` are only needed when your device will be establishing outgoing connections to other hosts on the network (perform active opens). If your device will only be accepting incoming connections then you *do not have to* set the `net.gatewayip` and the `net.netmask`.



Strangely, a lot of people hold a passionate belief that default gateway IP and the netmask are necessary always, even for incoming connections. This is not true!

The `net` object is usually initialized once on startup, like this:

```
sub on_sys_init
    ... some other stuff

    net.ip = "192.168.1.95" 'just an example! May not work on your
network!
    net.gatewayip = "192.168.1.1" 'just an example! May not work on your
network!
    net.netmask= "255.255.255.0" 'just an example! May not work on your
network!

    ...some other stuff
end sub
```



On a lot of networks the IP, gateway IP, and the netmask parameters of the hosts are configured automatically, through the use of a special protocol called "DHCP". The `net` object does not support dhcp directly by we provide a BASIC library that implements DHCP functionality.

One additional read-only property- the [net.mac](#)^[269]- can be used to extract the MAC address of your device. Your program cannot change the MAC address directly. The MAC is stored in the special configuration area of the EEPROM. Access to the EEPROM is provided by the [stor](#)^[380] object. To change the MAC address you need to rewrite the data in the EEPROM. For more details, see the [Stor Object](#)^[380] and [Stor. Base](#)^[381] Property topics.

Checking Ethernet Status

The [net.failure](#)^[271] read-only property tells you if the NIC is functioning properly.

The [net.linkstate](#)^[271] read-only property tells you if there is a live Ethernet cable plugged into the Ethernet port of your device, and, if yes, whether this is a 10BaseT or 100BaseT connection. The [on_net_link_change](#)^[271] event is generated each time the link status changes:

```
sub on_net_link_change

    if net.linkstate= PL_NET_LINKSTATE_NOLINK then
        'switch the RED LED on (just an example of what you could do)
        pat.play("RRRRRRRRRRRRRR", YES)
    else
        'switch the GREEN LED on
        pat.play("GGGGGGGGGGGGGG", YES)
    end if

end sub
```

Notice, that the `net.linkstate` always reflects current link status, not the link at the time of event generation.

Finally, there is a [on_net_overrun](#)^[272] event that is generated when internal RX buffer of NIC overflows.

Properties, Methods, Events

This section provides an alphabetical list of all properties, methods, and events of the `net` object.

.Mac R/O Property

Function:	Returns the MAC address of the Ethernet interface.
Type:	Dot-decimal string
Value Range:	Any valid MAC address, i.e. "0.1.2.3.4.5". Each device is preset with individual MAC address during production.
See Also:	---

Details

BASIC application cannot change MAC address directly. The MAC is stored in the EEPROM memory. This is the same memory used by the [stor](#)^[380] object. On power-up the MAC address is loaded from the EEPROM and programmed into the Ethernet controller of the device. Stor object provides a way to change MAC address in the EEPROM- see [Stor Object](#)^[380] for details.

.Ip Property

Function:	Sets/returns the IP address of the Ethernet interface of your device.
Type:	Dot-decimal string
Value Range:	Any valid IP address, such as "192.168.100.40". Default = "1.0.0.1"
See Also:	net.gatewayip , net.netmask

Details

If invalid IP addresses is specified it will be automatically corrected to the nearest valid IP-address. For example, "192.168.100.0" will be corrected to "192.168.100.1", "192.168" to "192.168.0.1", etc.

This property can only be written to when no socket is engaged in communications through the Ethernet interface, i.e. there is no socket for which [sock.statesimple](#) <> 0- PL_SSTS_CLOSED and [sock.currentinterface](#) = 1-PL_INTERFACE_ETHERNET.

.Netmask Property

Function:	Sets/returns the netmask of the Ethernet interface of your device.
Type:	String
Value Range:	Any valid netmask, such as "255.255.255.0". Default = "0.0.0.0"
See Also:	net.ip , net.gatewayip

Details

This property can only be written to when no socket is engaged in communications through the Ethernet interface, i.e. there is no socket for which [sock.statesimple](#) <> 0- PL_SSTS_CLOSED and [sock.currentinterface](#) = 1-PL_INTERFACE_ETHERNET.

.Gatewayip Property

Function:	Sets/returns the IP address of the default gateway for the Ethernet interface of your device.
Type:	String
Value Range:	Any valid IP address, such as "192.168.100.40". Default = "127.0.0.1"
See Also:	net.ip , net.netmask

Details

This property can only be written to when no socket is engaged in communications through the Ethernet interface, i.e. there is no socket for which [sock.statesimple](#)^[358] <> 0- PL_SSTS_CLOSED and [sock.currentinterface](#)^[33]= 1- PL_INTERFACE_ETHERNET.

.Failure R/O Property

Function:	Reports whether the Network Interface Controller (NIC) IC has failed.
Type:	Enum (no_yes, byte)
Value Range:	0- NO (default): No failure 1- YES: NIC failure
See Also:	---

Details

.Linkstate R/O Property

Function:	Returns current link status of the Ethernet port of the device.
Type:	Enum (pl_net_linkstate, byte)
Value Range:	0- PL_NET_LINKSTAT_NOLINK: No physical Ethernet link exists at the moment (the Ethernet port of the device is not connected to a hub). 1- PL_NET_LINKSTAT_10BASET: The Ethernet port of the device is linked to a hub (or directly to another device) at 10Mbit/sec. 2- PL_NET_LINKSTAT_100BASET: The Ethernet port of the device is linked to a hub (or directly to another device) at 100Mbit/sec.
See Also:	on_net_link_change ^[27]

Details

On_net_link_change Event

Function:	Generated when the state of the physical link of Ethernet port changes.
Declaration:	on_net_link_change

See Also: ---

Details

Multiple `on_net_link_change` events may be waiting in the event queue. This event does not "bring" with it new link state at the time of event generation. Current link state can be queried through the [net.linkstate](#)^[271] property.

On_net_overrun Event

Function: Generated when overflow occurs on the internal RX buffer of the Network Interface Controller (NIC) IC.

Declaration: `on_net_overrun`

See Also: ---

Details

Another `on_net_overrun` event is never generated until the previous one is processed. Notice, that this event signifies the overrun of the hardware RX buffer of the NIC itself. This has nothing to do with the overrun of RX buffers of individual sockets (see [on_sock_overrun](#)^[343] event).

Button Object



All external devices and boards offered by Tibbo feature a single button called "setup button". Our modules have a line to connect such a button externally. The setup button has a special system function: powering-up a Tibbo device with this button pressed (setup button line held LOW) causes the device to enter the serial upgrade mode. This mode is for uploading new [TiOS](#)^[7] (firmware) file into the device through the serial port.

The button is not doing anything system-related at other times, so it can be used by your Tibbo BASIC application -- hence, the button object. The object offers the following:

The [button.pressed](#)^[274] property returns current (immediate) state of the button.

The [on_button_pressed](#)^[273] event is generated when the button is pressed. The [on_button_released](#)^[273] event is generated when the button is released.

The [button.time](#)^[274] read-only property returns the time (in 0.5 intervals) elapsed since the button was last pressed or released. You can use this property, for instance, to separate button pressing into "short" and "long":

```

sub on_button_released
  'see how much time has elapsed

    if button.time>4 then
      'the button was pressed for a "long" time -- do one thing
    else
      'the button was pressed for a "short" time -- do another thing
    end if
end sub

```

The button does not require any pre-configuration and works always.

Note that the [on_button_pressed](#)^[273] and [on_button_released](#)^[273] events, as well as the [button.time](#)^[274] R/O property utilize "debouncing", which filters out very short transitions of the button state. The [button.pressed](#)^[274] R/O property, however, does not rely on debouncing and returns the immediate state of the button at the very moment the property is read.

On_button_pressed Event

Function:	Generated when the button on your device is pressed.
Declaration:	on_button_pressed
See Also:	Button.pressed ^[274]

Details

Multiple on_button_pressed events may be waiting in the event queue. You can check the time elapsed since the previous [on_button_released](#)^[273] event by reading the value of the [button.time](#)^[274] read-only property.

Note that the button object performs "debouncing" which rejects very brief transitions of the button state. This event will not be generated for such spurious transitions.

On_button_released Event

Function:	Generated when the button on your device is released.
Declaration:	on_button_released
See Also:	Button.pressed ^[274]

Details

Multiple on_button_released events may be waiting in the event queue. You can check the time elapsed since the previous [on_button_pressed](#)^[273] event by reading the value of the [button.time](#)^[274] read-only property.

Note that the button object performs "debouncing" which rejects very brief transitions of the button state. This event will not be generated for such spurious

transitions.

.Pressed R/O Property

Function:	Returns the current button state.
Type:	Enum (no_yes, byte)
Value Range:	0- NO: the button is not pressed. 1- YES: the button is pressed.
See Also:	---

Details

This property reflects an immediate state of the hardware at the very moment the property is read -- no "debouncing" performed. This is different from the [on_button_pressed](#)^[273] and [on_button_released](#)^[273] events, as well as the [button_time](#)^[274] R/O property, which all take debouncing into the account.

.Time R/O Property

Function:	Returns the time (in 0.5 second intervals) elapsed since the button was last pressed or released (whichever happened) later.
Type:	Byte
Value Range:	0-255
See Also:	---

Details

It only makes sense to read this property inside the [on_button_pressed](#)^[273] or [on_button_released](#)^[273] event handlers. Once the value of this property reaches 255 (127 seconds) it stays at 255 (there is no roll-over to 0). Elapsed time is not counted when the execution of your application is paused.

Sock Object



This is the sockets object. It allows you to maintain up to 16 simultaneous UDP or TCP ("normal" or HTTP) connections (actual number supported by the platform may be lower, due to memory constraints).

Very commonly, each connection is called a "socket". This is the term we will use as well. On other programming systems, sockets are often dynamic, created and destroyed as needed. With TiOS, you receive a preset number of sockets which have already been created for you, and just use them. A socket may be idle, but it will still be there.

Individual sockets have all the traditional settings you would expect to find, such as destination port number, protocol, etc. At the same time, their functionality goes significantly beyond what you usually find, and includes a lot of additional features that significantly lower the amount of code you need to write. For example, you can restrict incoming connections to your device, automatically filter out certain messages within the TCP data stream, etc.

The sockets object also implements webserver (HTTP) functionality. Each socket can carry a "normal" data connection or be in the HTTP mode.



Currently, the socket object can only access first 65534 bytes of each HTML file, even if the actual file is larger! Make sure that all HTML files in your project are not larger than 65534 bytes. This is not to be confused with the size of HTTP output generated by the file. A very large output can be generated by a small HTML file (due to dynamic data)- and that is OK. What's important is that the size of each HTML file in your project does not exceed 65534 bytes.

The sock object should not be confused with objects used to represent actual network interfaces, such as the [net](#)^[267] object which represents the Ethernet interface. The socket object is responsible for actual IP (TCP or UDP) communications -- it doesn't matter which interface these communications are effected through. Therefore, this is not the right place to look for a property such as 'IP address'. This is an attribute of a particular network interface.

Follows is the list of features offered by each socket of the sock object:

- Support for UDP, TCP and HTTP protocols (this is a submode of TCP).
- An extensive set of properties that define which hosts can connect to the socket, whether broadcasts are supported, which listening ports are associated with the socket, etc.
- Support for automatic processing of inband commands-- messages that are passed within the TCP data stream.
- Detailed socket state reporting with 30 different states supported!
- Fully asynchronous operation with separate "data arrival" and "data sent" events.
- Automatic data overrun detection on the RX buffer.
- Adjustable receive (RX), transmit (TX), and other buffer sizes for optimal RAM utilization.
- Buffer shorting feature for fast data exchange between the sock object and other objects (such as the [ser](#)^[224] object) that support standard Tibbo Basic data buffers.

Overview 3.5.1

This section covers the socket object in detail. Here you will find:

- [Anatomy of a socket](#)^[276]
- [Socket selection](#)^[276]

- [Handling Network Connection](#)^[277]
- [Sendind and Receiving Data](#)^[297]
- [Working With Inband Commands](#)^[309]
- [Using HTTP](#)^[314]

Anatomy of a Socket

A socket is composed of a send/receive logic that actually handles UDP, TCP (including HTTP) communications, and of 6 buffers. Each socket is capable of maintaining one connection with another node (host) on a network.

The socket object contains properties, methods and events which relate both to the buffers and the send/receive logic.

The *buffers* available are:

- The **RX** buffer, which stores data incoming from the host on the other side of a connection (this buffer doesn't have to be used for [HTTP connections](#)^[314]).
- The **TX** buffer, which stores data which is due for sending to the host on the other side of a connection (for HTTP connection, this buffer can store both the request and the reply).
- The **TX2** buffer, which is used internally, and only when [inband commands](#)^[309] are enabled.
- The **CMD** buffer, which is used to store incoming inband commands (messages). It is used only when inband commands are enabled.
- The **RPL** buffer, which is used to store outgoing inband replies (messages). It is used only when inband commands are enabled.
- The **VAR** buffer, which is used to store [HTTP](#)^[314] request string. It is needed only when the socket is in the HTTP mode.

Socket Selection

TiOS supports up to 16 sockets, but there may be platforms with less than 16 sockets available. You can obtain the number of sockets available for your platform using the [sock.numofsock](#)^[340] property.

Since there can be multiple sockets, you must state which socket are you referring to when changing properties or invoking methods. This is done using the [sock.num](#)^[340] property. For example:

```
sock.protocol = 1
```

Can you tell what socket the statement above applies to? Neither can the platform. Thus, the correct syntax would be:

```
sock.num = 0
sock.protocol = 1
```

Now the platform knows which socket you're working with. Once you have set the *socket selector* (using `sock.num`), every socket-specific method and property after that point is taken to refer to that socket. Thus:

```
sock.num = 0
sock.protocol = 1
sock.connectiontimeout = 10
sock.httpmode = 1 ' etc
```

The events for this object are not separate for each socket. An event such as [on_sock_data_arrival](#)^[342] serves all sockets on your platform. Thus, when an event handler for the socket object is entered, the socket selector is automatically switched to the socket number on which the event occurred:

```
sub on_sock_data_arrival
    dim s as string
    s = sock.getdata(255) ' Note that you did not have to specify any
    sock.num preceding this statement.
end sub
```

As a result of this automatic switching, when an event handler for a socket event terminates, the `sock.num` property retains its new value (nothing changes it back). You must take this into account when processing other event handlers which make use of a socket (and are not socket events). In other words, you should explicitly set the `sock.num` property whenever entering such an event handler, because the property might have been automatically changed prior to this event. To illustrate:

```
sub on_sys_init ' This is always the first event executed.
    sock.num = 0 ' Supposedly, this would make all subsequent
    properties and methods refer to this socket.
end sub

sub on_sock_data_arrival ' Then, supposing this event executes.
    dim s as string
    s = sock.getdata(255) ' However, this event happens on the second
    socket. So now sock.num = 1.
end sub

sub on_ser_data_arrival ' And then this serial port event executes.
    sock.txclear ' You meant to do this for sock.num = 0 (as specified
    at on\_sys\_init). But now sock.num was changed to 1! Oops...
end sub
```

To recap, only one of two things may change the current `sock.num`: **(1)** manual change or **(2)** a socket event. You cannot assume the number has remained unchanged if you set it somewhere else (because a socket event might have happened since).

Handling Network Connections

The whole purpose of socket's existence is to engage in network connections with other network hosts. Each socket can maintain a single connection using a UDP/IP or TCP/IP *transport* protocol. Which of the two is used is defined by the [sock.protocol](#)^[345] property.

Sockets are also capable of working with HTTP. HTTP is not a transport protocol, rather, it is based on the TCP. Therefore, when your socket is using TCP it may be for "plain data transmission" or for HTTP.

TCP connection basics

What is TCP

The TCP is the most widely used transmission protocol. It is the backbone of all Internet traffic. The idea behind the TCP is to provide two communicating points (we can call them host A and host B) with a reliable, stream-oriented data link. "Stream-oriented" means that neither the host A, nor the host B have to worry about how the data travels across the connection. A just puts the stream of bytes in and B receives exactly the same stream of bytes on its side. It is the responsibility of the TCP to split the data into packets for transmission through the network, retransmit lost packets, make sure there are no data overruns, etc.

The TCP is strictly a "point-to-point" protocol: only two parties can engage in a connection and no third party can "join in".

TCP connections

Before any data can be transmitted one of the hosts has to establish a connection to another host. This is similar to placing a telephone call: one of the parties has to call the other end.

The host that takes initiative to establish a connection is said to be opening an "outgoing connection" or "performing an active open". This is like dialing a telephone number of the desired party, only the number is the IP address of another host.

The host that accepts the "call" is said to be accepting an "incoming connection" or "performing a passive open". This is similar to picking up the phone when it starts ringing.

Once connection has been established, both parties can "say something" (send data) at any time and the TCP will make sure that all data sent on one end arrives to the other end.

TCP connections are expected to be closed (terminated) properly- there is a special exchange of messages between the host to let each other know that connection is being terminated. This is called "graceful disconnect". There is also a "reset" (abort) which is much simpler and is akin to hanging up abruptly. Finally, there is a "discard" way to end the connection in which the host simply "forgets" that there was a connection.

The TCP connection can be closed purposefully, or it can [timeout](#).

A TCP connection in progress is fully defined by 4 parameters: IP address and the port number on host A and the IP address and port number on host B. When the host is performing an active open, it has to "dial" not just the IP address of the target host, but also the port number on this host. Ports are not physical- they are just logical subdivisions of the IP address (65536 ports per IP). If the IP is a telephone number of the whole office then the port is an extension. The "calling" host is also calling not just from its IP address but also from specific port.

UDP "connection" basics

What is UDP

In many aspects, the UDP protocol is the opposite of the TCP protocol. Whereas the TCP provides a reliable, stream-oriented transport, the UDP offers a way to send data as separate packets or "UDP datagrams". This is similar to "paging" (as in, sending a message with a "pager" -- remember those?). What is sent is a packet containing some data. There is no guarantee that the other side will receive

the packet and the sender won't know whether the packet was received or not.

Each UDP datagram lives its own life and you are responsible for dividing your data into chunks of reasonable size and sending it in separate datagrams. There is no connection establishment or termination on UDP- the datagram can be sent instantly, without any preparation.

Unlike TCP, the UDP is not point-to-point. For example, several hosts can send the datagrams to the same socket of your device- and all of them can be accepted- a situation that is impossible with TCP. There is also an option to broadcast the datagram to all hosts connected to the same network segment- something that is also impossible with TCP.

And now to UDP "connections"...

This said, it should come as a bit of a surprise that we will now turn 180 degrees and start talking about UDP "connections". Didn't we just say that there is no such thing? Well, yes and no. On the physical network there is, indeed, no such thing. However, on our socket object level we have deliberately made UDP communications look more like TCP connections. And, since the UDP "connection" is nothing more than our sock object's abstraction, we use the word "connection" in quotation marks.

We consider an active open to have been performed when our host has sent the first UDP datagram to the destination. A passive open is when we have received the first incoming UDP datagram from another host (provided that this datagram was accepted- more on that in [Handling Incoming Connections](#)^[279]).

There is no graceful disconnect for UDP connections. The connection can only be discarded (our host "forgets" about it). The UDP connection can also [timeout](#)^[280].

Accepting Incoming Connections

Master switch for incoming connections

It is possible to globally enable or disable acceptance of incoming connections on all sockets, regardless of the setup of individual sockets. This is done using the [sock.inconenabledmaster](#)^[337] property. By default, this property is set to 1- YES.

Defining who can connect to your socket

The [sock.inconmode](#)^[338] ("incoming connections mode") property allows you do define whether incoming connections will be accepted and, if yes, from who. By default (0- PL_SOCK_INCONMODE_NONE), incoming connections are not allowed at all. For TCP this means that incoming connection requests will be rejected. For UDP, incoming datagrams will simply be ignored.

If you don't mind to accept an incoming connection from any host/port then set the [sock.inconmode](#)^[338]= 3- PL_SOCK_INCONMODE_ANY_IP_ANY_PORT. This way, whoever wants to connect to your socket will be able to do so as long as this party is using correct transport protocol (you define this through the [sock.protocol](#)^[345] property) and is connecting to the right port number (more on this below).

If you are only interested in accepting connections from a particular host on the network then set [sock.inconmode](#)^[338]= 2- PL_SOCK_INCONMODE_SPECIFIC_IP_ANY_PORT. This way, only the host with IP matching the one defined by the [sock.targetip](#)^[359] property will be able to connect to your socket.

You can restrict things further and demand that not only the IP of the other host must match the one you set in [sock.targetip](#)^[359] property, but also the port from

which the connection is being originated must match the port defined by the [sock.targetport](#)^[36] property. To achieve this, set the [sock.inconmode](#)^[338]= 1-PL_SOCK_INCONMODE_SPECIFIC_IPPORT.

Here is an example of how to only accept incoming TCP connections from host 192.168.100.40 and port 1000:

```
sock.protocol= PL_SOCK_PROTOCOL_TCP
sock.inconmode= PL_SOCK_INCONMODE_SPECIFIC_IPPORT
sock.targetip= "192.168.1.40"
sock.targetport= 1000
```

The sock object rejects an incoming connection by sending out a reset TCP packet. This way, the other host is instantly notified of the rejection. There is an exception to this -- see [Socket Behavior in the HTTP Mode](#)^[318].

Listening ports

Ports on which your socket will accept an incoming connection are called "listening ports". These are defined by two properties: the [sock.localportlist](#)^[339] and the [sock.httpportlist](#)^[335]. Notice that both properties are of string type, so each one can accept a list of ports.

For example, to accept a normal data connection either on port 1001, port 2000, or port 3000, set the `sock.localportlist= "1001,2000,3000"`. Once the connection is in progress, you can check which of the socket's local ports is actually engaged in this connection. This is done through the [sock.localport](#)^[338] read-only property.

For UDP connections, the `sock.localportlist` is all there is. For TCP, which can be used for "plain vanilla data connections" or for HTTP, you have another property-`sock.httpportlist`. To be accepted by your socket, an incoming TCP connection has to target either one of the ports on the `sock.localportlist`, or one of the ports on the `sock.httpportlist`. The socket will automatically switch into the HTTP mode if the connection is accepted on one of the ports from the `sock.httpportlist`.

Here is an example:

```
sock.localportlist = "1001,2000"
sock.httpportlist = "80"
```

The above means that any incoming TCP connection that targets either port 1001 or port 2000 will be interpreted as a regular data connection. If connection target port 80 it will be accepted as an HTTP connection.

And what if the same port is listed both under the `sock.localportlist` and `sock.httpportlist`?

```
sock.localportlist = "1001,2000,80"
sock.httpportlist = "80"
```

In this example, if there is an incoming connection targeting "our" port 80 and the protocol is TCP then the mode will be HTTP- the `sock.httpportlist` has priority over `sock.localportlist`. Of course, for UDP the `sock.httpportlist` won't matter since the

HTTP is only possible on TCP!

Setting allowed interfaces

The sock object is interface-independent and supports communications over more than one interface. The [sock.allowedinterfaces](#)^[327] property defines the list of interfaces on which the current socket will accept incoming connections. The list is different and depends on the platform. Each platform section of the manual contains a topic named "Platform-dependent Programming Information". You will find the list of available interfaces inside this section, under the "Available network interfaces" heading.

The [sock.allowedinterfaces](#)^[327] property stored the string that lists all interfaces that the socket will listen on:

```
sock.allowedinterfaces = "NET,WLN" 'listen on Ethernet and Wi-Fi  
interfaces
```

Connections accepted even when the VM is paused

Once the socket has been setup it will accept an incoming connection even when the VM is paused (for example, has stopped on your breakpoint). All communications are handled by the [master process](#)^[7], so the socket does not need the VM to accept an incoming connection (or, for that matter, receive and send the data).

Accepting UDP broadcasts

UDP datagrams can be sent as broadcasts. Broadcast, instead of specifying a particular network host as a destination, targets a group of hosts on the network.



The sock object supports link-level broadcasts. Such broadcast packets have their destination MAC address set to 255.255.255.255.255. Link-level broadcasts are received by all network hosts connected to the current network segment. Link-level broadcasts cannot penetrate routers, bridges, etc.

For the socket to accept incoming UDP broadcasts, the [sock.acceptbcst](#)^[327] property must be set to 1- YES. In every other aspect working with incoming broadcast UDP datagrams is like working with regular incoming datagrams.

Understanding TCP Reconnects

The sock object has a unique feature- support for "reconnects". To simplify the explanation, let us start from a description of a real-life problem first.

When reconnects save the day

Supposing, you have a host on the network that is engaged in a TCP connection with one of the sockets of the socket object. The data is sent across this connection from the host to our socket. For a while, everything is fine and then the host momentarily loses power and reboots. Our socket doesn't know anything about this- from its point of view, the TCP connection is still OK. Just because no data is arriving from the host does not mean that there is a problem!

Meanwhile, the host reboots and attempts to establish a new connection to the socket- and gets rejected! This is because the socket thinks it is already engaged in a TCP connection- the one that has been left hanging since the host went down! This stagnant connection will remain in place until it times out- if timeouts are enabled at all through the [sock.connectiontout](#)^[330] property.

Reconnects are a nifty way out of this situation. You enable reconnects through the [sock.reconmode](#)^[345] (reconnection mode) property. For the above example, you typically set the `sock.reconmode= 2- PL_SOCK_RECONMODE_2`. This mode means, that if the socket is already engaged in a connection, and then there is an incoming connection attempt that originated from the same host (and any port), then the socket will forget everything about the original connection and accept the new one.

For our example case, this is the solution- after rebooting the host tries to establish a new connection to our socket. The socket then "realizes" that this new connection is being attempted from the same host as the connection already in progress, discards the original connection and accepts the new one!

Reconnects must target the same port and interface!

Even when the socket has more than one listening port (i.e. [sock.localportlist](#)^[339]= "1000,1001") the reconnect will only be accepted if it targets the same local port of the socket as the one already engaged in the current connection being "replaced". In other words, to be successful, the reconnect must target the port that is currently returned by the [sock.localport](#)^[338] read-only property.

The interface must also be the same. The host can't make an original connection through Ethernet, and then reconnect through Wi-Fi.

Which mode to choose?

As you can see, the `sock.reconmode` property gives you several "strictness levels" of dealing with reconnects. Which one to choose? Let us explain why the choice of `sock.reconmode= 2- PL_SOCK_RECONMODE_2` is the most common. Typically, when the host is establishing an outgoing connection it does so from the ever changing port number. Basically, there is a "pool" of ports for this purpose and each new connection the host needs to establish will be made from the next port in the pool.

Each time the host connects to our socket the port on the host could be different. This is why it makes sense to accept reconnects from the same IP but any port. Disadvantage? Any connection originating from this host will essentially be treated as the same and the only connection!

Some programs (few!) establish connections from a specific, preset port. This may be done for a variety of reasons- no time to go into this here- just let us say, sometimes this is the case. If you are dealing with such a case then you can safely set `sock.reconmode= 1- PL_SOCK_RECONMODE_1`. This way, reconnects will only be accepted from the same IP and the same port as the original connection.

Total promiscuity -- mode 3!

Finally, there is a mode (3- `PL_SOCK_RECONMODE_3`) when the socket will accept

a reconnect from any host or port. Basically, this means, that whatever connection is in progress, it will be interrupted and replaced by any other incoming connection.

Do not use reconnects for HTTP sockets!

Reconnects and HTTP do not play nicely together. When you request an HTML page, several simultaneous HTTP requests may be generated (one for the page itself, several -- for pictures on this page, etc.). All these requests will use a separate TCP socket, so multiple sockets will be opened (almost) at the same time. Now, what will happen if even just one of your application's "HTML" sockets has reconnects enabled? This single socket will intercept all HTML requests. So, if loading the HTML page needed 3 separate requests and TCP sessions, this socket will get them all -- and each next session opening will discard the previous one. Result won't be pretty!

Understanding UDP Reconnects and Port Switchover

For UDP "connections", there is also such a thing as reconnects. Due to a very different nature of UDP (compared to TCP), reconnects for UDP must be explained separately. Additionally, we need to introduce something called "port switchover".

Port switchover explained

With TCP, each side of the connection uses a single port both to send and receive data. With UDP, this doesn't have to be the case. The `sock` object, when it is engaged in a connection, can receive the data from one port but send the data to a different port!

When port switchover is disabled, the socket always addresses its outgoing UDP datagrams to the port, specified by the `sock.targetport`^[360] property. When port switchover is enabled, the socket will address its outgoing datagrams to the port from which the most recent incoming datagram was received!

Notice, that we did not say anything about the IP switchover- so far we have only discussed ports.

UDP reconnects

Just like with TCP, the `sock.reconmode`^[345] property defines, for UDP, what kind of incoming UDP datagram will be able to make the socket forget about its previous "connection" and switch to the new one. You have two choices: either define that reconnects are only accepted from a specific IP but any port or choose reconnects to be accepted from any IP and any port. Combine this with two options for port switchover and you have four combinations- four options for the `sock.reconmode` property. All this is best understood on the example.

Example: PL_SOCK_RECONMODE_0

Setup:

```
sock.protocol= PL_SOCK_PROTOCOL_UDP 'we are dealing with UDP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IPPORT
sock.reconmode= PL_SOCK_RECONMODE_0 'reconnects accepted from the same IP,
any port, port switchover off
sock.localportlist= "3000"
sock.targetport= "900"
```

Two hosts are sending us datagrams and here is how the socket will react:

Incoming datagram

192.168.100.40:1000 sends UDP datagram to the port 3000 of our device

192.168.100.44:1001 sends UDP datagram to the port 3000 of our device

192.168.100.40:1100 sends UDP datagram to the port 3000 of our device

Socket reaction

Datagram accepted, UDP "connection" is now in progress, the socket will be sending all further outgoing datagrams to 192.168.100.40:900.

Datagram ignored- it came from a different IP. The socket will still be sending its outgoing datagrams to 192.168.100.40:900.

Reconnection accepted (so datagram is accepted), but the socket will still be sending all further outgoing datagrams to 192.168.100.40:900 (because port switchover is off).

Example: PL_SOCK_RECONMODE_1

Setup:

```
sock.protocol= PL_SOCK_PROTOCOL_UDP 'we are dealing with UDP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IPPORT
sock.reconmode= PL_SOCK_RECONMODE_1 'reconnects accepted from any IP/port,
port switchover off'
sock.localportlist= "3000"
sock.targetport= "900"
```

Two hosts are sending us datagrams and here is how the socket will react:

Incoming datagram

192.168.100.40:1000 sends UDP datagram to the port 3000 of our device

192.168.100.44:1001 sends UDP datagram to the port 3000 of our device

Socket reaction

Datagram accepted, UDP "connection" is now in progress, the socket will be sending all further outgoing datagrams to 192.168.100.40:900.

Reconnection accepted, the socket will be sending all further outgoing datagrams to 192.168.100.44:900.

Example: PL_SOCK_RECONMODE_2

Setup:

```
sock.protocol= PL_SOCK_PROTOCOL_UDP 'we are dealing with UDP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IPPORT
sock.reconmode= PL_SOCK_RECONMODE_2 'reconnects accepted from the same IP,
any port, port switchover on'
sock.localportlist= "3000"
sock.targetport= "900"
```

Two hosts are sending us datagrams and here is how the socket will react:

Incoming datagram

192.168.100.40:1000 sends UDP datagram to the port 3000 of our device

192.168.100.44:1001 sends UDP datagram to the port 3000 of our device

192.168.100.40:1100 sends UDP datagram to the port 3000 of our device

Socket reaction

Datagram accepted, UDP "connection" is now in progress, the socket will be sending all further outgoing datagrams to 192.168.100.40:1000.

Datagram ignored- it came from a different IP. The socket will still be sending its outgoing datagrams to 192.168.100.40:1000.

Reconnection accepted, the socket will be sending all further outgoing datagrams to 192.168.100.40:1100.

Example: PL_SOCK_RECONMODE_3

Setup:

```
sock.protocol= PL_SOCK_PROTOCOL_UDP 'we are dealing with UDP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IPPORT
sock.reconmode= PL_SOCK_RECONMODE_3 'reconnects accepted from any IP/port,
port switchover on
sock.localportlist= "3000"
sock.targetport= "900"
```

Now, two hosts are sending us datagrams and here is how the socket will react:

Incoming datagram

192.168.100.40:1000 sends UDP datagram to the port 3000 of our device

192.168.100.44:1001 sends UDP datagram to the port 3000 of our device

Socket reaction

Datagram accepted, UDP "connection" is now in progress, the socket will be sending all further outgoing datagrams to 192.168.100.40:1000.

Reconnection accepted, the socket will be sending all further outgoing datagrams to 192.168.100.44:1001.

Incoming Connections on Multiple Sockets

So far, we have been talking about all kinds of incoming connections applied to a single socket. The fact is, the sock object supports up to 16 sockets which all can have a different setup. Just because an incoming connection is rejected or ignored on one socket does not mean that it won't be connected on another!

When the network packet is received by the sock object the latter attempts to find a socket for which this newly arrived packet is acceptable.

First, the sock object checks if the packet can be considered a part of any existing connection or a reconnection attempt for an existing connection. All sockets that are currently engaged in a connection are checked, starting from socket 0, then sock 1, 2, and up to `sock.numofsock`^{[34]-1}.

If it turns out that some socket can accept the packet as a part of current connection or an acceptable reconnection attempt then the search is over and the packet is "pronounced" to belong to this socket.

If it turns out that no socket currently engaged in a connection can accept the packet then the socket object checks all currently idle sockets to see if any of these sockets can accept this packet as a new incoming connection. Again, all idle sockets are checked, starting from socket 0, and up to `sock.numofsock`-1.

For TCP, a packet that can start such a new connection is a special "SYN" packet. For UDP, any incoming datagram that can be accepted by the socket (which depends on the socket setup) can start the new connection.

If the packet cannot be construed as a part of any existing connection, re-connect, or new incoming connection then this packet is discarded.

Example

Supposing, we have the following setup:

```

sock.num= 0
sock.protocol= PL_SOCK_PROTOCOL_TCP
sock.inconmode= PL_SOCK_INCONMODE_SPECIFIC_IP_ANY_PORT
sock.targetip="192.168.100.40"
sock.reconmode= PL_SOCK_RECONMODE_2
sock.localportlist= "1001"

sock.num=1
sock.protocol= PL_SOCK_PROTOCOL_TCP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IPPORT
sock.localportlist= "1001"

sock.num=2
sock.protocol= PL_SOCK_PROTOCOL_UDP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IPPORT
sock.localportlist= "2000"

```

Here is a sample sequence of incoming connections and how the setup above would handle them:

Incoming datagram

Socket reaction

Incoming TCP connection to port 1001 from host 192.168.100.40:29600	Connection will be accepted on socket 0 since this socket lists 1001 as one of the listening ports and this incoming connection request is from "correct" host.
Incoming TCP connection to port 1001 from host 192.168.100.40:29601	This will be taken as a reconnect on socket 0: this incoming connection is from the same host as the previous one and it targets the same port 1001.
Incoming TCP connection to port 1001 from host 192.168.100.41:900	Connection will be accepted on socket 1, because socket 0 is already engaged in a connection and this new connection request cannot be interpreted as a reconnect (different host).
Incoming UDP datagram to port 2000 from host 192.168.100.40:320	This datagram will be accepted on socket 2 and "connection" will be opened.

Establishing Outgoing Connections

Performing active opens

Now that we know how to setup the sockets to accept incoming TCP connections and UDP "connections" we move on to learning about establishing connections of our own, or, as is often said, performing "active opens".

Establishing an outgoing connection is always an explicit action- you use the [sock.connect](#)^[330] method to attempt to do this. Once you do this the socket will try to perform an active open to the IP and port specified by the [sock.targetip](#)^[359] and [sock.targetport](#)^[360] properties. There is also a [sock.targetinterface](#)^[359] property -- this one defines which network interface the new connection will be passing through.



As you can see, the `sock.targetip` and `sock.targetport` properties perform a double-duty: for incoming connections they define (if required by the [sock.incomemode](#)^[338]) who will be able to connect to the socket. For outgoing connections this pair defines IP and host to which the socket will attempt to connect.

Notice, that your `sock.connect` invocation will only work if you do this while the socket is in the "closed" state (see [Checking Connection Status](#)^[292]).

Active opens for TCP

Once you tell a "TCP" socket to connect, the socket will do the following:

- Resolve the IP address of the target using the ARP protocol.
- Attempt to engage the target in a standard TCP connection sequence (SYN-SYN-ACK).
- Connection will be either established, or this will fail. Your program has a way to monitor this- see [Checking Connection Status](#)^[292].



If what we've just said doesn't ring any bell for you, don't worry. These, indeed, are very technical things and you don't have to understand them fully to be able to use the sock object.

"Active opens" for UDP

When you tell an "UDP" socket to connect, the latter will just resolve the IP address of the target and consider "connection" established.

Do not forget: connection handling is fully asynchronous!

Keep in mind that the sock object handles communications asynchronously. When the VM executes the [sock.connect](#)^[330] method it does not mean that the connection is established by the time the VM gets to the next program statement. Executing `sock.connect` merely instructs the master process to start establishing the connection (more on master process and the VM in the [System Components](#)^[74] topic). Connection establishment can take some time and your application doesn't have to wait for that to complete. [Checking Connection Status](#)^[292] topic explains how to find out actual connection status at any time (see [sock.state](#)^[355] and [sock.statesimple](#)^[358] R/O properties).



Asynchronous nature of the sock object has some interesting implications. [More On the Socket's Asynchronous Nature](#)^[294] topic contains important information on the subject, so make sure you read it!

Sending UDP broadcasts

How to send UDP broadcasts

UDP datagrams can be sent as broadcasts. Broadcast, instead of specifying a particular network host as a destination, targets a group of hosts on the network.



The sock object supports link-level broadcasts. Such broadcast packets have their destination MAC address set to 255.255.255.255. Link-level broadcasts are received by all network hosts connected to the current network segment. Link-level broadcasts cannot penetrate routers, bridges, etc.

To make the socket send its outgoing UDP datagrams as broadcasts, set the [sock.targetbcast](#)^[359] property to 1- YES:

```
...
sock.targetbcast= YES
sock.setdata("ABC") 'this is explained in 'Working With Buffers' section
sock.send 'this is explained in 'Working With Buffers' section
sock.connect 'broadcast UDP datagram with string 'ABC' will be sent out at
this point
...
```

There is one difference to grasp regarding the socket that is sending its outgoing packets as broadcasts: no incoming UDP packet that would normally be interpreted as a re-connect will cause the socket to "switchover" to the source IP-address of the packet.

Let's evaluate two examples.

Example 1: regular UDP communications

Here is a sample setup for the UDP socket:

```
sock.protocol= PL_SOCK_PROTOCOL_UDP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IP_ANY_PORT
sock.reconmode= PL_SOCK_RECONMODE_3
sock.targetip= "192.168.100.40"
sock.targetport= 1000
sock.localportlist= "2000"
sock.setdata("ABC")
sock.send
sock.connect
```

And here is a hypothetical sequence of events:

Incoming/outgoing datagram

Datagram with contents "ABC" sent to 192.168.100.40:1000 as soon as sock.connect method is invoked

Incoming UDP datagram from 192.168.100.41:20.

Socket sends out another datagram- this time to 192.168.100.41:20!

Comment

We now have the UDP "connection" with 192.168.100.40:1000

This will be taken as a reconnect- the socket is now engaged in a connection with 192.168.100.41:20.

Complete switchover happened- the socket is now transmitting data to the IP and port of the sender of the previous incoming datagram

Example 2: regular UDP communications

Here is another setup for the UDP socket:

```
sock.protocol= PL_SOCK_PROTOCOL_UDP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IP_ANY_PORT
sock.reconmode= PL_SOCK_RECONMODE_3
sock.targetbcast= YES
sock.targetport=1000
sock.localportlist= "2000"
sock.setdata("ABC")
sock.send
sock.connect
```

Sequence of events:

Incoming/outgoing datagram	Comment
Datagram with contents "ABC" sent as broadcast to all stations on the network segment	We now have the UDP "connection" with... err... everybody on the segment
Incoming UDP datagram from 192.168.100.41:20.	This is still a reconnect, but the socket won't switch over to the IP-address of the sender. Only port switchover will take place!
Socket sends out another datagram- still as a broadcast but this time to port 20.	Port switchover happened because it is allowed by the sock.reconmode ^[345] .

Closing Connections

Passive TCP connection termination

When your socket is engaged in a TCP connection with another host on the network, this host may choose to terminate the connection. This can be done through a "graceful disconnect" sequence ("FIN-ACK-FIN-ACK") or through a reset ("RST" packet).

In both cases the socket will handle connection closing automatically, without any help from your Tibbo BASIC program or the VM. In fact, the socket will accept connection termination even when the VM is stopped (for example, it was paused on your breakpoint). All communications are handled by the [master process](#)^[7], so the socket does not need the VM to terminate the connection.

There is one intricate detail in the connection termination process that you will have to understand clearly. When the VM is not running, the socket can [accept an incoming connection](#)^[279] (active open) and can accept an active close. However, the socket won't be able to accept another (next) incoming connection until the VM has had a chance to run and execute the [on_sock_event](#)^[343] event handler (see [Checking Connection Status](#)^[292]).

Here is why. In many cases there is a need to perform certain actions (like, maybe, clear some buffers, initialize variables, etc.) after the previous connection ends and before the new one begins. Not letting the socket accept next connection before your program has a chance to respond to connection termination is a way to achieve this!

Actively closing TCP connections

A TCP connection can be closed, reset (aborted), or discarded using three different methods- [sock.close](#)^[328], [sock.reset](#)^[348], and [sock.discard](#)^[331].

The recommended (and polite!) way of closing a TCP connection is through the `sock.close` method. This will initiate a proper closing sequence ("FIN-ACK-FIN-ACK"), known as a "graceful disconnect".

Connection reset is slightly more "rude"- your socket will simply tell the other end that "it is all over" (by sending the "RST" packet).

Finally, the `sock.discard` method simply makes your socket forget about the connection- the other side is not notified at all.

Just like with connection establishment, you can monitor the progress of connection termination- see [Checking Connection Status](#)^[292].



Notice, that depending on the socket state at the moment of `sock.close` method invocation, the socket may need to resort to a simple connection closing option- reset or discard. Similarly, when you use the `sock.reset`, it may sometimes result in discard. For more information read the [`sock.close`](#)^[328], [`sock.reset`](#)^[348], and [`sock.discard`](#)^[331] topics.

Actively closing UDP connections

For UDP, no matter what method you use to close the "connection", result will still be as if the `sock.discard` was invoked. This is because in reality there is no such thing as a proper UDP connection termination so simply "forgetting" about the connection is the only option the socket has.

Do not forget: connection handling is fully asynchronous!

Keep in mind that the `sock` object handles communications asynchronously. When the VM executes the [`sock.close`](#)^[328] ([`sock.reset`](#)^[348], [`sock.discard`](#)^[331]) method it does not mean that the connection is done with by the time the VM gets to the next program statement. Executing these methods merely instructs the master process to terminate the connection. Connection termination can take some time and your application doesn't have to wait for that to complete. [Checking Connection Status](#)^[292] topic explains how to find out actual connection status at any time (see [`sock.state`](#)^[355] and [`sock.statesimple`](#)^[358] read-only properties).



Asynchronous nature of the `sock` object has some interesting implications. [More On the Socket's Asynchronous Nature](#)^[294] topic contains important information on the subject, so make sure you read it!

Socket re-use after connection closing

When the socket connection terminates, the socket is ready to accept another incoming connection or establish a new outgoing connection (if so configured) -- with one little caveat! There is a special built-in mechanism that ensures that your application has a chance to react after the previous connection terminates and before the next one is established.

For example, you might need to clean some buffers before each new incoming TCP connection. Naturally, you want this to happen before the new connection is actually accepted.

Typically, your program achieves this by executing code placed in the [on_sock_event](#)^[343] event handler (this event is explained in [Checking Connection Status](#)^[292] topic). The socket will not be able to engage in the new connection until the `on_sock_event` has a chance to execute. There is an interesting example on this in the [More On the Socket's Asynchronous Nature](#)^[294] topic.

Connection timeouts

The [`sock.connectiontout`](#)^[330] provides a way to automatically terminate a connection across which no data was exchanged for a predefined period of time. For TCP, `reset` (abort) is used, while UDP "connections" are simply discarded. Connection

timeout is a useful way to exit "hanged" connections (this happens a lot with TCP on large networks).

The [sock.toutcounter](#)^[360] R/O property informs your application of the time passed since the data was last exchanged across the connection. Each time there is some data sent or received the sock.toutcounter is reset to zero. The property increments at 0.5 second intervals while no data is moving through this socket.

If the sock.connectiontout is not at 0, the sock.toutcounter increments until it reaches the value of the sock.connectiontout and the connection is terminated. The sock.toutcounter then stays at the value of sock.connectiontout.

If the sock.connectiontout is at 0, the maximum value that the sock.toutcounter can reach is 1. That is, the sock.toutcounter will be at 0 after the data exchange, and at 1 if at least 0.5 seconds have passed since the last data exchange.

Normally, HTTP connections close automatically

There is one case where your socket will perform an active graceful disconnect without you using the sock.close method. This is the case when the socket is running in the TCP-HTTP mode.

In [Accepting Incoming Connections](#)^[279] we have already explained that the socket automatically switches into the HTTP mode if a TCP connection is accepted on one of the ports from the [sock.httpportlist](#)^[335] list. The socket can also be switched into the HTTP mode programmatically, through the [sock.httpmode](#)^[334] property.

Default HTTP functionality requires that the TCP connection is closed once the HTTP server has finished sending out its "response" (i.e. HTML page or another file that has been requested). In this situation the socket won't need the sock.close from your program- the connection will be terminated automatically. In fact, when the socket is in the HTTP mode, your [sock.close](#)^[328], [sock.reset](#)^[348], and [sock.discard](#)^[331] will simply be ignored. There is a [sock.httpnoclose](#)^[335] property alters the standard socket behavior in the HTTP mode. Set this property to 1- YES and the connection will be kept opened even after the socket has sent all of the HTTP reply out.

Just like in all other cases, a new connection on the socket won't be accepted until your program has had a chance to respond- this was explained above.

Checking Connection Status

Simplified and detailed socket states

The sock object features several properties that provide complete information on what the socket is doing, which IP and port it is engaged in the connection with, etc.

The most important of all this data is the socket state. Two state groups are supported- "simplified" and "detailed". Simplified state tells you, generally, what condition the socket is in. Detailed state additionally tells you how this condition came to be.

For example, the "simplified" PL_SSTS_CLOSED state means that connection is closed (socket is idle). It doesn't tell you, however, why it is closed. "Detailed" state PL_SST_CL_ARESET_CMD tells you that connection is closed because there was an active close as a result of the [sock.close](#)^[328] method invocation from your program!

On_sock_event and sock.event, sock.events simple read-only properties

Each time the socket state changes an [on_sock_event](#)^[343] event is generated. Unlike many other events, this one can be generated again and again before the on_sock_event handler is invoked, so event queue can contain multiple such events. A separate event is generated for each new state the socket enters.

The on_sock_event "brings" two arguments with it -- one is called "newstate", another one -- "newstatesimple". These arguments contains the state of the socket at the moment when the on_sock_event was generated.

Newstate and newstatesimple arguments have been introduced in Tibbo Basic **V2.0**. Before, their role was played by two read-only properties -- [sock.event](#)^[332] and [sock.eventsimple](#)^[332]. These properties are no longer available.

Here is one example of how on_sock_event can be used. Supposing, when connection is established we need to open the serial port and when it is no longer established close the serial port and clear the data from its TX and RX buffers:

```
sub on_sock_event(newstate as pl_sock_state,newstatesimple as
pl_sock_state_simple)
    if newstatesimple= PL_SSTS_EST then
        ser.enabled= YES 'connection has been established- open port
    else
        if ser.enabled= YES then 'connection is NOT established? Close and
clear the port if it is opened!
            ser.enabled= NO
            ser.txclear
            ser.rxclear
        end if
    end if
end sub
```

Sock.state and sock.statesimple read-only properties

There is also a pair of read-only properties- [sock.state](#)^[355] and [sock.statesimple](#)^[358] that allows you to check current socket state (as opposed to the state that was at the moment of on_sock_event generation).

Here is an example of how you can use this. Supposing, you want to "play" an LED pattern that depends on the current state of the socket. Here is how you do this:

```
sub on_pat
'this event is generated each time a pattern finishes playing
    select case sock.statesimple
        case PL_SSTS_EST: 'connection is established- Green LED on
            pat.set("GGGGGGGGGGGGGGGG",NO)
        case PL_SSTS_ARP: 'ARP in progress- Green LED blinks
            pat.set("G-G-G-G-G-G-G-",NO)
        case PL_SSTS_PO: 'connection is being established- green LED goes
'blink-blink-blink'
            pat.set("-----G-G-G-",NO)
        case PL_SSTS_AO: 'connection is being established- green LED goes
'blink-blink-blink'
            pat.set("-----G-G-G-",NO)
        case else: 'connection is closed or being closed- green LED is
'blink-blink'
            pat.set("-----G-G-",NO)
    end select
end sub
```

Understanding who you are talking to

Whenever the socket is engaged in the connection you can check the parameters of the other side through three read-only properties- `sock.remotemac`^[348], `sock.remoteip`^[347], and `sock.remoteport`^[348]. For UDP, you can also check if the datagram you have received was sent to your device exclusively or it was a broadcast- the `sock.bcast`^[328] will tell you that. For TCP, you can additionally check if the socket is in the "regular data" or HTTP mode- just check the `sock.httpmode`^[334] property.

There is an intricate detail to understand about the `sock.remotemac`, `sock.remoteip`, `sock.remoteport`, and `sock.bcast` properties when you are using the UDP protocol.

With UDP, your socket may be accepting datagrams from several different hosts. As will be explained in [Receiving Data in UDP Mode](#)^[303], the most common way to handle the incoming data is through the `on_sock_data_arrival`^[342] event. You will get one such event for each UDP datagram that the socket will receive. If you check the `sock.remotemac`, `sock.remoteip`, `sock.remoteport`, or `sock.bcast` from within the `on_sock_data_arrival` event handler you will get the sender's data for the UDP datagram *currently* being processed.

On the contrary, using `sock.remotemac`, `sock.remoteip`, `sock.remoteport`, or `sock.bcast` property outside of the `on_sock_data_arrival` event handler will give you the data for the *most recent* UDP datagram received into the RX buffer of the socket. This is not the same as the *next* UDP datagram to be extracted from the RX buffer and processed by your application!

Checking current interface

Starting from **V1.2**, the `sock` object provides an additional property -- `-- sock.currentinterface`^[331] (**not supported** by the [EM202/200 \(-EV\), DS202](#)^[134] platform) -- which tells you which network interface the network connection is going through.

More On the Socket's Asynchronous Nature

In [Establishing Outgoing Connections](#)^[287] and [Closing Connections](#)^[290] topics we have already touched on the subject of the `sock` object's asynchronous nature. This topic offers further details on what that means for your application.

Executing `sock.connect`^[330], `sock.close`^[328], `sock.reset`^[348], or `sock.discard`^[331] method does not mean that your connection gets established or terminated by the time your program reaches the next statement. Executing these statements merely instructs the Master Process what to do with the connection (more on the Master Process in the [System Components](#)^[7] topic). Connection establishment/termination can take some time and your application doesn't have to wait for that to complete. [Checking Connection Status](#)^[292] topic explained how to find out actual connection status at any time (see `sock.state`^[355] and `sock.statesimple`^[358] read-only properties).

There are certain situations when your program has to take the above into account. Here is one example. Supposing, we want to know the MAC address of a remote device to which we are establishing an outgoing connection. Naturally, we can do it this way:

```
'Correct code -- on startup we order the connection to be established and
in the on_sock_event event handler we record the MAC address of the
'other side'

Sub On_sys_init
    ...
    sock.targetip="192.168.100.40"      'we prepare for the connection
    sock.targetport=2000
    sock.connect                      'and now we connect
End Sub

'-----
'-----
Sub On_sock_event(newstate As pl_sock_state,newstatesimple As
pl_sock_state_simple)
    Dim s As String
    If newstatesimple=PL_SSTS_EST Then
        'OK, so connection is established, let's get this MAC!
        s=sock.remotemac
    End If
End Sub
```

The above is a good example of event-driven programming. Sometimes, however, you need to establish a connection and "follow-up" on it in the same event handler. So, how do we do this? Here is a simple, and WRONG code:

```
'!!! BAD EXAMPLE !!!

Dim s As String
...
...
sock.targetip="192.168.100.40"      'we prepare for the connection
sock.targetport=2000
sock.connect                      'and now we connect
s=sock.remotemac                  'and now we try to check the MAC. WRONG!
Connection may not be established yet!

...  
...
```

And here is the correct way to handle this. For clarity, this example assumes that connection will definitely be established.

```
'Correct, but simplified example (we do not handle possible connection
failure).

Dim s As String
...
...
sock.targetip="192.168.100.40"      'we prepare for the connection
sock.targetport=2000
sock.connect                      'and now we connect
While sock.statesimple<>PL_SSTS_EST 'we wait here until the connection is
actually established
Wend
s=sock.remotemac                  'Get the MAC!  
...
```

Here is even more interesting example. Supposing, you want to close and

reestablish a TCP connection right within the same event handler. Here is a wrong way of doing this:

```
'!!! BAD EXAMPLE !!! -- this just won't work!
...
...
sock.close
sock.connect
...'
```

You see, executing `sock.close` doesn't really close the connection -- it only issues the instruction (to the Master Process) to close the connection. So, by the time program execution gets to the `sock.connect` method your previous connection is still on!

Correct way is to wait for the connection to actually be closed before executing `sock.connect`. Here is another example -- not quite correct either -- but closer to the truth.

```
'!!! 'BETTER' CODING !!! -- still not totally OK!
...
...
sock.close
  While sock.statesimple<>PL_SSTS_CLOSED 'here we wait for the connection
  to be closed
    Wend
  sock.connect
...'
```

OK, this is better. One final correction and the code is complete. In the [Closing Connections](#) topic ("Socket re-use after connection closing" section) we have already explained that the OS makes sure that the `on_sock_event` has a chance to execute after the old connection is closed and before the new one is established. In the above example both `sock.close` and `sock.connect` are in the same event handler -- the `on_sock_event` won't squeeze in between them unless you use the [doevents](#) statement! Here is the correct code:

```
'100% CORRECT!
...
...
sock.close
  While sock.statesimple<>PL_SSTS_CLOSED 'here we wait for the connection
  to be closed
    Wend
  doevents ' Absolutely essential for this particular case!
  sock.connect
...'
```

Notice how `doevents` is placed after the while-wend loop. It is absolutely essential that you do it this way! Of course, now that you have at least one `doevents` in the event handler you might as well add `doevents` in all "places of waiting" -- just to let other events execute sooner.

```
'Even better code!
...
sock.close
  While sock.statesimple<>PL_SSTS_CLOSED 'here we wait for the connection
  to be closed
    doevents 'Not necessary but useful -- lets other events execute
  Wend
  doevents ' Absolutely essential for this particular case!
sock.connect
...
```



You have to place doevents after the while-wend loop and you have to do this even if you don't actually have a handler for the [on_sock_event](#) [343] event in your application!

Sending and Receiving data

Once a network connection has been established the socket is ready to send and receive the data. This is done through two buffers- the TX buffer and the RX buffer. Read on and you will know how to allocate memory for buffers, use them, handle overruns, and perform other tasks related to sending and receiving of data.

Allocating Memory for Buffers

Each buffer has a certain size, i.e, a memory capacity. This capacity is allocated upon request from your program. When the device initially boots, no memory is allocated to buffers at all.

Memory for buffers is allocated in pages. A *page* is 256 bytes of memory. Allocating memory for a buffer is a two-step process: first you have to request for a specific allocation (a number of pages) and then you have to perform the actual allocation.

For the socket object to be able to send and receive the data, you have to give its TX and RX buffers some memory. This is done through the [sock.txbuffrq](#) [362] and [sock.rxbuffrq](#) [350] methods.

The allocation method ([sys.buffalloc](#) [217]) applies to all buffers previously specified, in one fell swoop.

Hence:

```
dim in, out as byte
out = sock.txbuffrq(10) ' Requesting 10 pages for the TX buffer. Out will
then contain how many can actually be allocated.

in = sock.rxbuffrq(7) ' Requesting 7 pages for the RX buffer. Will return
number of pages which can actually be allocated.

' ... Allocation requests for other buffers...

sys.buffalloc ' Performs actual memory allocation, as per previous
requests.
```

Actual memory allocation takes up to 100ms, so it is usually done just once, on boot, for all required buffers. If you do not require some buffer, you may choose not to allocate any memory to it. In effect, it will be disabled.

You may not always get the full amount of memory you have requested. Memory is not an infinite resource, and if you have already requested (and received) allocations for 95% of the memory for your platform, your next request will get up to 5% of memory, even if you requested for 10%.

There is a small overhead for each buffer. Meaning, not 100% of the memory allocated to a buffer is actually available for use. 16 bytes of each buffer are reserved for variables needed to administer this buffer, such as various pointers etc.

Thus, if we requested (and received) a buffer with 2 pages ($256 * 2 = 512$), we actually have 496 bytes in which to store data ($512 - 16$).



If you are *changing* the size of any buffer for a socket using `sys.buffalloc`, and this socket is not closed ([sock.statesimple](#)^[358] is not `PL_SSTS_CLOSED`), the socket will be automatically closed. Whatever connection you had in progress will be discarded. The socket will not be closed if its buffer sizes remain unchanged.

Using Buffers in TCP Mode

Once you have allocated memory for the TX and RX buffers you can start sending and receiving data through them. Since TCP is a stream-oriented protocol this is what buffers store- a stream of data being sent and received, without any separation into individual packets. Even for the outgoing data, you have no control over how it will be split into packets for transmission over the network.

Sending Data

Sending data a two-step process. First, you put the data in the TX buffer using the [sock.setdata](#)^[353] method, and then you perform the actual sending (commit the data) using the [sock.send](#)^[353] method. For example:

```
sock.setdata ("Foo") ' Placed our data in the TX buffer - not being sent
out yet.
' ... more code...
sock.setdata ("Bar") ' Added even more data to the TX buffer, waiting to
be sent.
sock.send ' Now data will actually start going out. Data sent will be
'FooBar'.
```

Since this is a two-step process, you may gradually fill the buffer to capacity, and only then send its contents.



TiOS features *non-blocking operation*. This means that on `sock.send`, for example, the program does not halt and wait until the data is completely sent. In fact, execution resumes immediately, even before the first byte goes out. Your program will not freeze just because you ordered it to send a large chunk of data.

The data can be stored in the TX buffer at any time but it will only be sent out if and when the [network connection](#)^[277] is established. Storing the data in the TX buffer won't cause the socket to establish any connection automatically.

Receiving Data

Receiving data is a one-step process. To extract the data from the RX buffer, use the [sock.getdata](#)^[33] method. Data may only be extracted once from the buffer. Once extracted, it is no longer in the buffer. For example:

```
dim whatigot as string
whatigot = sock.getdata(255)
```

The string whatigot now contains up to 255 bytes of data which came from the RX buffer of the socket.

Discussion of TCP data RXing continues in [Receiving Data in TCP mode](#)^[30].

Using Buffers in UDP Mode

UDP is a packet-based protocol (as opposed to TCP, which is stream-based). In UDP you usually care what data belongs to what datagram (packet). The sockets object gives you complete control over the individual datagrams you receive and transmit.

Sending and receiving UDP data is still effected through the TX and RX buffers. The difference is that the subdivision into datagrams is preserved within the buffers.

Each datagram in the buffer has its own header:

- For the TX buffer, headers contain datagram length as well as the destination MAC, IP, port, and broadcast flag (indicating whether to send the datagram as a broadcast).
- For the RX buffer, headers contain datagram length plus the sender's MAC, IP, port, and broadcast flag (indicating whether the datagram is a broadcast).

Sending Data

Just like with TCP, sending data through the TX buffer in UDP mode is a two-step process; first you put the data in the buffer using the [sock.setdata](#)^[35] method, and then you close a datagram and perform the actual sending (commit the data) using the [sock.send](#)^[35] method.

The datagrams will never be mixed with one another. Once you invoke `sock.send`, the datagram is closed and sent (as soon as possible). Any new data added to the TX buffer will belong to a new datagram. For example:

```
sock.setdata ("Foo") ' Placed our data in the tx buffer - not being sent
out yet.

' ... more code...

sock.setdata ("Bar") ' Added even more data to the same datagram, waiting
to be sent.
sock.send ' A datagram containing 'FooBar' will now be closed and
committed for sending.
sock.setdata ("Baf") ' This new data will go into a new datagram.
sock.send ' Closes the datagram with only 'Baf' in it and commits it for
sending.
sock.send ' sends an empty UDP datagram!
```

Notice that in the example above we were able to send out an empty datagram by using `sock.send` without `sock.setdata`!

Keep in mind that there is a limitation for the maximum length of data in the UDP datagram- 1536 bytes.

Receiving Data

Receiving data in UDP mode requires you to be within an event handler for incoming socket data, or to explicitly move to the next UDP datagram in the buffer.

To extract the data from a buffer, use the [sock.getdata](#)^[338] method. This method only accesses a single datagram on the buffer, unless you use the [sock.nextpacket](#)^[339] method. If you have several incoming datagrams waiting, you will have to process them one by one, moving from one to the next. This is good because this way you know where one datagram ends and another one begins.

Here is an example:

```
sub on_sock_data_arrival
    dim whatigot as string
    whatigot = sock.getdata(255) 'will only extract the contents of a
    single datagram. Reenter the on_sock_data_arrival to get the next datagram
end sub
```

Data may only be extracted once from a buffer. Once extracted, it is no longer in the buffer. Discussion of UDP data RXing continues in [Receiving Data in UDP mode](#)^[303].

TX and RX Buffer Memory Status

You cannot effectively use a buffer without knowing what is its status. Is it overflowing? Can you add more data? etc. Thus, each of the socket buffers has certain properties which allow you to monitor it:

The RX buffer

You can check the total capacity of the buffer with the [sock.rxbuffsize](#)^[351] property. You can also find out how much data the RX buffer currently contains with the [sock.rxlen](#)^[352] property. From these two data, you can easily deduce how much free space you have in the RX buffer -- even though this isn't such a useful datum (that's one of the reasons there is no explicit property for it).

Note that `sock.rxlen` returns the *gross* current size of data in the RX buffer. In TCP mode, this is equivalent to the actual amount of data in the buffer. However, in UDP mode, this value includes the headers preceding each datagram within the RX buffer -- the amount of actual data in the buffer is smaller than that. A separate property -- [sock.rxpacketlen](#)^[353] returns the length of actual data in the UDP datagram you are currently processing.

Sometimes you need to clear the RX buffer without actually extracting the data. In such cases the [sock.rxclear](#)^[363] comes in handy.

The TX buffer

Similarly to the RX buffer, the TX buffer also has a [sock.txbuffsize](#)^[363] property which lets you discover its capacity.

The TX buffer has two "data length" properties: [sock.txlen](#)^[364] and [sock.newtxlen](#)^[339]. The `txlen` property returns the amount of *committed* data waiting to be sent from the buffer (you commit the data by using the [sock.send](#)^[353] method). The `newtxlen` property returns the amount of data which has entered the buffer, but has not yet

been committed for sending. For UDP, this is basically the length of UDP datagram being created.

The TX buffer also has a [sock.txfree](#)^[363] property, which directly tells you how much space is left in it. This does not take into account uncommitted data in the buffer -- actual free space is `sock.txfree - sock.newtxlen!`



`ser.txlen + ser.txfree = ser.txbuffsize.`

When you want to clear the TX buffer without sending anything, use the [sock.txclear](#)^[363] method. Notice, however, that this will only work when the network connection is closed ([sock.statesimple](#)^[358]= PL_SSTS_CLOSED).

An example illustrating the difference between `sock.txlen` and `sock.newtxlen`:

```
sub on_sys_init
    dim x,y as word ' declare variables
    sock.rxbuffrq(1) ' Request one page for the RX buffer.
    sock.txbuffrq(5) ' Request 5 pages for the TX buffer (which we will use).
    sys.buffalloc ' Actually allocate the buffers.

    sock.setdata("foofoo") ' Set some data to send.
    sock.setdata("bar") ' Some more data to send.
    sock.send ' Start sending the data (commit).
    sock.setdata("baz") ' Some more data to send.
    x = sock.txlen ' Check total amount of data in the TX buffer.
    y = sock.newtxlen ' Check length of data not yet committed.

    end sub 'Set up a breakpoint HERE.
```

Don't step through the code. The sending is fast -- by the time you reach `x` and `y` by stepping one line at a time, the buffer will be empty and `x` and `y` will be 0. Set a breakpoint at the end of the code, and then check the values for the variables (by using the [watch](#)^[33]).

Receiving Data in TCP Mode

We have already explained that RX and TX buffers operate differently in the [TCP](#)^[298] and [UDP](#)^[299] mode. This section explains how to receive data when the TCP protocol is used by the socket.

In a typical system, there is a constant need to handle an inflow of data. A simple approach is to use polling. You just poll the buffer in a loop and see if it contains any fresh data, and when fresh data is available, you do something with it. This would look like this:

```
sub on_sys_init
    while sock.rxlen = 0
    wend ' basically keeps executing again and again as long as sock.rxlen = 0
    s = sock.getdata(255) ' once loop is exited, it means data has arrived. We extract it.

    end sub
```

This approach will work, but it will forever keep you in a specific event handler

(such as [on_sys_init](#)^[220]) and other events will never get a chance to execute. This is an example of *blocking code* which could cause a system to freeze. Of course, you can use the [doevents](#)^[82] statement, but generally we recommend you to avoid this blocking approach.

Since our platform is event-driven, you should use events to tell you when new data is available. There is an [on_sock_data_arrival](#)^[342] event which is generated whenever there is data in the RX buffer:

```
sub on_sock_data_arrival
    dim s as string
    s = sock.getdata(255) ' Extract the data -- but in a non-blocking way.
    ' .... code to process data ....
end sub
```

The `on_sock_data_arrival` event is generated whenever there is data in the RX buffer, but only once. There are never two `on_sock_data_arrival` events (for the same socket) waiting in the queue. The next event is only generated after the previous one has completed processing, if and when there is any data available in the RX buffer.

This means that when handling this event, you don't have to get *all* the data in the RX buffer. You can simply handle a chunk of data and once you leave the event handler, a new event of the same type will be generated.

Here is a correct example of handling arriving socket data through the `on_sock_data_arrival` event. This example implements a data loopback -- whatever is received by the socket is immediately sent back out.

```
dim rx_tcp_packet_len as word 'to keep the size of the next incoming TCP
packet
sub on_sys_init
end sub
```

We want to handle this loopback as efficiently as possible, but we must not overrun the TX buffer. Therefore, we cannot simply copy all arriving data from the RX buffer into the TX buffer. We need to check how much free space is available in the TX buffer. The first line of this code implements just that: `sock.getdata` takes as much data from the RX buffer as possible, but not more than `sock.txfree` (the available room in the TX buffer). The second line just sends the data.



Actually, this call will handle no more than 255 bytes in one pass. Even though we seemingly copy the data directly from the RX buffer to the TX buffer, this is done via a temporary string variable automatically created for this purpose. In this platform, string variables cannot exceed 255 bytes.

Receiving Data in UDP Mode

In the [previous section](#)^[301] we have already explained how to handle data reception when the socket is in the TCP mode. This section provides explanation for receiving data with UDP.

Using `on_sock_data_arrival` event

With UDP, you still (typically) process incoming data basing on the [on_sock_data_arrival](#)^[342] event. Two differences apply:

- Each time the `on_sock_data_arrival` event handler is entered you only get to process a single UDP datagram waiting in the RX buffer. Unless you use a special method (see below), you won't be able to get the data from the next datagram, even if this datagram is already available in the RX buffer.
- If, while within the `on_sock_data_arrival` event handler, you don't read out entire contents of the "current" datagram and exit the event handler, then the unread portion of the datagram is discarded.

Here is an example: supposing, two datagrams are waiting in the RX buffer and their contents are "ABC" and "123". The following code will then execute twice:

```
sub on_sock_data_arrival
dim s as string(2)
s = sock.getdata(255) ' will get 2 bytes as this is the capacity of s
end sub
```

Since `s` has a maximum capacity of 2 the first time `s=sock.getdata(255)` executes you will get "AB". When the event handler is exited the rest of the first UDP datagram will be discarded, so next time you will get "12"!

Using `sock.nextpacket` method

Using `on_sock_data_arrival` event is a preferred method of handling an incoming data stream. Still, in selected cases you may need to process RX data in a loop, without leaving the event handler.

The [`sock.nextpacket`](#)^[339] method exists for just such a case. The result of this method execution is equivalent to exiting/(re)entering the `on_sock_data_arrival` event handler: the unread portion of the previous UDP datagram is discarded and we move to the next UDP datagram (if any).

Here is a code example where we handle all incoming UDP data without exiting event handler:

```

dim s as string
...
11:
while sock.rxlen = 0
    doevents 'good practice: let other events execute while we are
waiting
wend

sock.nextpacket 'Now we know that we do have data, 'move to' the next UDP
datagram. This is like entering the on_sock_data_arrival.
s = sock.getdata(255) 'get data
goto 11

...

```

Sending TCP and UDP Data

In the previous sections, we have explained how to handle an incoming stream of data. You could say it was incoming-data driven. Sometimes you need just the opposite -- you need to perform operations based on the sending of data.

Sending data with `on_sock_data_sent` event

Supposing that in a certain system, you need to send out a long string of data when a button is pressed. A simple code for this would look like this:

```

sub on_button_pressed
    sock.setdata("This is a long string waiting to be sent. Send me
already!")
    sock.send
end sub

```

The code above *would* work, but *only* if at the moment of code execution the necessary amount of free space was available in the TX buffer (otherwise the data would get truncated). So, obviously, you need to make sure that the TX buffer has the necessary amount of free space before sending. A simple polling solution would look like this:

```

sub on_button_pressed
    dim s as string
    s = "This is a long string waiting to be sent. Send me already!"
    while sock.txfree < len(s)
        wend
    sock.setdata(s)
    sock.send
end sub

```

Again, this is not so good, as it would block other event handlers. So, instead of doing that, we would employ a code that uses [on_sock_data_sent](#)^[342]:

```

dim s as string
s = "This is a long string waiting to be sent. Send me already!"

sub on_button_pressed
    sock.notifySent(sock.txBufferSize-len(s)) ' causes the
on_sock_data_sent event to fire when the TX buffer has space for our
string
end sub

sub on_sock_data_sent
    sock.setdata(s) ' put data in TX buffer
    sock.send ' start sending it.
end sub

```

When we press the button, [on_button_pressed](#)^[273] event is generated, so now the system knows we have a string to send. Using [sock.notifySent](#)^[340] we make the system fire the [on_ser_data_sent](#) event when the necessary amount of free space becomes available. This event will only be fired once -- and will be fired immediately if there is already enough available space.

Within the event handler for this event, we put the data in the TX buffer and start sending it.



Amount of data that will trigger [on_sock_data_sent](#) does not include uncommitted data in the TX buffer.

UDP datagrams are generated as you create them

In [Using Buffers in TCP Mode](#)^[299] we have already explained that for UDP you have a complete control over how the data you are sending is divided into the UDP datagrams. Each time you use the [sock.send](#) method you draw the boundary between the datagrams- the previous one is "closed" and the new one "begins". You can even send out an empty UDP datagram by executing [sock.send](#)^[353] without using the [sock.setdata](#)^[353] first.

Correctly responding to the sender of each UDP datagram

With UDP, your socket may be receiving UDP datagrams from several different hosts. When the [on_sock_data_arrival](#) event handler is entered (see [Receiving Data in UDP mode](#)^[303]) the following properties automatically reflect the source of the current datagram- [sock.remoteMac](#)^[348], [sock.remoteIP](#)^[347], and [sock.remoteReport](#)^[348]. Additionally, the [sock.bcast](#)^[328] property will tell you whether the datagram was a regular or a broadcast one (this material has already been covered in [Checking Connection Status](#)^[292]).

Additionally, any datagram that was generated and sent from within the [on_sock_data_arrival](#) event handler will be sent to the sender of the datagram for processing of which the [on_sock_data_arrival](#) event handler was entered.

The point is that if you are sending out a datagram from within the [on_sock_data_arrival](#) event handler you are automatically replying to the sender of the datagram being processed. The following example sends back "GOT DATAGRAM FROM xxx.xxx.xxx.xxx" string in response to any datagram received by the socket:

```

sub on_sock_data_arrival
    sock.setdata("GOT DATAGRAM FROM " + sock.remoteip)
    sock.send
end sub

```

For the above example, even if several hosts send the datagrams to the socket at the same time each one of these hosts will get a correct reply back!

Now consider this example: each time the button is pressed the same message is generated:

```

sub on_button_pressed
    sock.setdata("GOT DATAGRAM FROM " + sock.remoteip)
    sock.send
end sub

```

The difference is that when you press the button the datagram will be sent to the destination, from which the most recent incoming UDP datagram was received (and accepted) by the socket!

"Split Packet" Mode of TCP Data Processing

Though our customer's feedback we have learned that sometimes it may be necessary to know the size of individual TCP packets. For example, as it turns out, some data encryption methods work on a packet level. Erase the border between TCP packets -- and you can't decrypt the data.

Introduced in Tibbo Basic V2.0, new [sock.splittcpackets](#)^[355] property and [on_sock_tcp_packet_arrival](#)^[344] event allow you to process incoming TCP data packet by packet. To achieve this, set the `sock.splittcpacket= 1- YES`. After that, the `on_sock_tcp_packet_arrival` will be generated for each incoming TCP packet carrying any new data (i.e., not a retransmission data). Here is an example of use:

```

dim rx_tcp_packet_len as word 'to keep the size of the next incoming TCP
packet

sub on_sys_init
    ...
    rx_tcp_packet_len=0
    sock.splittcpackets=YES
end sub

sub on_sock_tcp_packet_arrival(len as word)
    rx_tcp_packet_len=len 'we get the size of the next packet we are going
    to process
end sub

sub on_sock_data_arrival
    dim s as string
    'take exactly the contents of one packet (we assume that it can fit in
    the string!)
    s=sock.getdata(rx_tcp_packet_len)
    rx_tcp_packet_len=0 'very important!
end sub

```

Naturally, you may also need to control the size of outgoing TCP packets. This is done in a different way. With [sock.splittcpackets](#)^[355]= 1- YES, when you put some

data into the TX buffer and execute [sock.send](#)^[353], the socket won't send this data out unless entire contents of the TX buffer can be sent out in a single TCP packet. Here is an example of how you can use that:

```
sock.splittcpackets=YES
...
...
sock.setdata("ABCDEFGH1234567890")
sock.send
while sock.txlen>0
    doevents
wend
```

Notice that this method has its disadvantage -- your data throughput is diminished because your program is seeking an acknowledgement for each TCP packet being sent. On the other hand, this is a bulletproof way of making sure that the outgoing TCP packet contains exactly the data you intended.

Be careful not to try to send more data than the RX buffer size on the other end. Since in this mode the socket won't send that data unless it can send all of it, your TCP connection will get stuck!

Also, attempting to send the packet with size exceeding the "maximum segment size" (MSS) as specified by the other end will lead to data fragmentation! The socket will never send any TCP packet with the amount of data exceeding MSS.

Handling Buffer Overruns

Handling RX buffer overruns

The [on_sock_overrun event](#)^[343] is generated when an RX buffer overrun has occurred. It means the data has been arriving to the RX buffer faster than you were handling it and that some data got lost.

This event is generated just once, no matter how much data is lost. A new event will be generated only after exiting the handler for the previous one.

Normally, data overruns will not occur when the TCP transport protocol is used. This is because the TCP is intelligent enough to regulate the flow of data between the sender and the receiver and, hence, avoid overruns. The UDP protocol does not have a "flow control" mechanism and RX buffer overruns can and will happen.

Typically, the user of your system wants to know when an overrun has occurred. For example, you could blink a red LED when this happens.

```
sub on_ser_overrun
    pat.play("R-R-R-R")
end sub
```

Are TX buffer overruns possible?

TX buffer overruns are not possible. The socket won't let you overload its TX buffer. If you try to add more data to the TX buffer than the free space in the buffer allows to store then the data you are adding will be truncated.

See [Sending Data](#)^[304] for explanation on how to TX data correctly.

Redirecting Buffers

The following example appeared under [Receiving Data in TCP Mode](#)^[307]:

```
sub on_sock_data_arrival
    sock.setdata(sock.getdata(sock.txfree) )
    sock.send
end sub
```

This example shows how to send all data incoming to the RX buffer out from the TX buffer, in just two lines of code. However fast, this technique still passes all data through your BASIC code, even though you are not processing (altering, sampling) it in any way.

A much more efficient and advanced way to do this would be using a technique called *buffer redirection* (buffer shorting). With buffer shorting, instead of receiving the data into the RX buffer of your socket, you are receiving it directly into the TX buffer of another object which is supposed to send out this data. This can be a socket (same or different one), a serial port object, etc.

To use buffer shorting, you invoke the [sock.redir](#)^[346] method and specify the buffer to which the data is to be redirected. Once this is done, the on_sock_data_arrival event won't be generated at all, because the data will be going directly to the TX buffer that you have specified. As soon as the data enters this buffer, it will be automatically committed for sending. Here is an example:

```
sub on_sys_init
    sock.num=0
    sock.redir(PL_REDIR_SOCK0) ' This is a loopback for socket 0.
end sub
```

The performance advantage here is enormous, due to two factors: first, you are not handling the data programmatically, so the VM isn't involved at all. And second, the data being received is received directly into the TX buffer from which it is transmitted, so there is less copying in memory.

Of course you cannot do anything at all with this data -- you are just pumping it through. However, very often this is precisely what is needed! Additionally, you can still process [inband commands/replies](#)^[309] (messages).

To stop redirection, you can use `sock.redir(0)`, which means "receive data into the RX buffer of the socket in a normal fashion".

Redirection and UDP

[Using Buffers in UDP Mode](#)^[299] explained that the `sock` object preserves buffer boundaries when storing UDP datagrams in RX and TX buffer. To achieve this, the `sock` object uses special datagram headers that are also stored in these buffers. This means that the buffers contain not only data, but also an additional "service" information.

When an RX buffer of a "UDP socket" (i.e. a socket running in the UDP mode) is redirected to a TX buffer of another UDP socket, datagram boundaries are preserved. The receiving socket will still send out the data subdivided into the original datagrams.

When an RX buffer of a UDP socket is redirected to a TX buffer of a TCP socket or serial port, the service information is removed and datagram boundaries are

dissolved. To the receiving TCP socket or serial port, the data appears to be a stream.

Sinking Data

Sometimes it is desirable to ignore all incoming data while still maintaining the connection opened. The [sock.sinkdata](#)^[354] property allows you to do just that.

Set the `sock.sinkdata` to 1- YES, and all incoming data will be automatically discarded. This means that the [on_sock_data_arrival](#)^[342] event will not be generated, reading [sock.rxlen](#)^[352] will always be returning zero, and so on. No data will be reaching its destination even in case of [buffer redirection](#)^[308]. [Inband commands](#)^[309], however, will still be extracted from the incoming data stream and processed. [Sock.connectiontout](#)^[330] and [sock.toutcounter](#)^[360] will work correctly as well.

Working With Inband Commands

Inband commands (replies) are messages embedded within the TCP data stream. Each inband message has a specific formatting that allows the socket to recognize it in the data stream being received.

Inband messages have many uses. For example, in a network-to-serial converter, you typically pass all serial data through a single TCP connection. At the same time, you often need to send some control commands to the device, i.e. for setup, etc. This can be done in two ways: through a separate network connection ("out-of-band" respective to the main data connection) or by embedding those commands inside the serial data stream ("inband" way). The second method is sometimes better, since very often you want to avoid or cannot have another network connection to your device.

Of course, you could just write a BASIC code that would be separating inband commands from the data but this would affect the performance (data throughput) of your device considerably. The `sock` object natively supports inband commands to avoid this performance penalty. In fact, inband commands will work even when the [buffer redirection](#)^[308] is enabled!



Inband messages are only possible with TCP transport protocol. You cannot use inband messages with UDP!

Inband Message Format

Inband message passing is enabled through the [sock.inbandcommands](#)^[337] property.

Each inband message has to start with a special escape character whose ASCII code is specified by the [sock.escchar](#)^[332] property. The next character after the escape character can have any ASCII code except for the code of the escape character itself.

Following that is the body of the inband message. The last character in the message is a so-called end character, specified by the [sock.endchar](#)^[331] property. It signals the end of the inband message and return to the "regular" data.

There are no specific limitations on how long the inband message can be. The length is only limited by how much space you allocate for the [CMD and RPL buffers](#)^[310] that store incoming and outgoing inband messages.

And what if the data stream itself contains a character(s) with the ASCII code of the escape character you have set? Wouldn't this confuse the socket into thinking that this is the beginning of inband command? To avoid this situation, the *data*

character with code of escape character is transmitted as two identical characters with the same ASCII code.

Example: supposing you have the following setup:

```
...
sock.inbandcommands= YES
sock.escchar='$'
sock.endchar='%'
```

Now, you have the following data stream coming into the socket:

ABCD\$#inband commqand%EFG\$\$123

The socket will interpret this stream as including one inband command: "\$#inband command%". Regular data, placed into the RX buffer of the socket will be "ABCDEFG\$123". The first '\$' character is interpreted as the beginning of the inband message because this character is followed by some other character ('#'). The second occurrence of the '\$' character is interpreted as data, since this character is followed by another 'S' character. Resulting data stream contains only a single '\$' character- the socket takes care of removing the second one automatically.

When sending data from the TX buffer, the socket also automatically doubles all data characters with the ASCII code of the escape character. So, if you want to send this string: "Outbound\$!" what will actually be sent is:

Outbound\$\$!



Inband messages are not our invention. Many programs, such as the HyperTerminal, treat the character with code 255 as an escape character.

Inband-related Buffers (CMD, RPL, and TX2)

Three buffers are required for inband message processing. On startup, these buffers are not allocated any memory, so you have to do it if you are planning to send and receive inband messages.

- The CMD buffer- used to store incoming inband messages (we call them "inband commands"). Use the [sock.cmdbuffrq](#)^[32] method to allocate memory for this buffer. Usually, inband commands are not very long so allocating a minimum space of 1 page is typically sufficient.
- The RPL buffer- used to store outgoing inband messages (we call them "inband replies"). Use the [sock.rplbuffrq](#)^[34] method to allocate memory for this buffer. again, inband commands are usually not very long so allocating a minimum space of 1 page will probably be sufficient.
- The TX2 buffer- used internally by the socket when inband commands are enabled. You don't have to do anything with this buffer other than allocate memory for it. We recommend allocating as much space as you did for the TX

buffer. Allocation is requested through the [sock.tx2buffrq](#)^[36] method.

Actual memory allocation is done through the [sys.buffalloc](#)^[217] method which applies to all buffers previously specified. Here is an example:

```
dim b1, b2, b3, b4, b5 as byte
...
'setup for other objects, sockets, etc.

b1= sock.txbuffrq(5) 'you need this buffer to send regular data
b2= sock.rxbuffrq(5) 'you need this buffer to receive regular data
b3= sock.cmdbuffrq(1) 'buffer for incoming inband commands
b4= sock.rplbuffrq(1) 'buffer for outgoing inband replies
b5= sock.tx2buffrq(5) 'same buffer size as for the TX buffer

....
sys.buffalloc ' Performs actual memory allocation, as per previous
               requests
```

Actual memory allocation takes up to 100ms, so it is usually done just once, on boot, for all required buffers.

You may not always get the full amount of memory you have requested. Memory is not an infinite resource, and if you have already requested (and received) allocations for 95% of the memory for your platform, your next request will get up to 5% of memory, even if you requested for 10%.

There is a small overhead for each buffer. Meaning, not 100% of the memory allocated to a buffer is actually available for use. 16 bytes of each buffer are reserved for variables needed to administer this buffer, such as various pointers etc.

Thus, if we requested (and received) a buffer with 2 pages ($256 * 2 = 512$), we actually have 496 bytes in which to store data ($512 - 16$).



If you are *changing* the size of any buffer for a socket using `sys.buffalloc`, and this socket is not closed ([sock.statesimple](#)^[358] is not `PL_SSTS_CLOSED`), the socket will be automatically closed. Whatever connection you had in progress will be discarded. The socket will not be closed if its buffer sizes remain unchanged.

Processing Inband Commands

What goes into the CMD buffer

All incoming inband commands are stored into the CMD buffer. At any given time the buffer may contain more than one command. Each inband message in the buffer already has its escape character and the character after the escape character removed. The end character, however, is not removed and can be used by your program to separate inband messages from each other.

Here is an example. Supposing, you have the following setup:

```
...
sock.inbandcommands= YES
sock.escapechar='@'
sock.endchar='&'
...
```

Here is a sample data stream:

Data@command1&Moredata@!command2&Evenmoredata

This incoming data stream will have the following effect:

- The RX buffer will receive this data: "DataMoredataEvenmoredata".
- The CMD buffer will receive: "ommand1&command2&"

Notice, that the first inband command is missing the first character- this is because when the inband command is being processed both its escape character and the character following the escape character are removed. End characters of both inband commands are preserved so you can tell where each one ends.

Extracting the data from the CMD buffer

Extracting data from the CMD buffer is similar to extracting data from the RX buffer. A dedicated method- [sock.getinband](#)^[334]- does the job. This method is just like the [sock.getdata](#)^[333], minus the maxinplen argument. Total amount of data in the CMD buffer can be checked through the [sock.cmdlen](#)^[329] property.

Once extracted, the data is no longer in the buffer. You can use the [sock.cmdlen](#)^[329] property to check how much data is waiting in the CMD buffer. There is no dedicated property to tell you the buffer capacity- just remember what the [sock.cmdbuffrq](#)^[329] method returned if you have to know this!

The [on_sock_inband](#)^[343] event is generated whenever there is some data in the CMD buffer, but only once. There are never two on_sock_inband events waiting in the queue. The next event is only generated after the previous one has completed processing, if and when there is any data available in the CMD buffer.

Inband commands only appear in the CMD buffer in their entirety. That is, if the buffer was previously empty and you get the on_sock_inband event then you are guaranteed that the buffer will contain a full command (or several full commands).

Here is an example:

```

sub on_sock_inband
    dim s as string 'we will keep the data from the CMD buffer here
    dim s2 as string 'this will keep individual inband commands
    dim x as byte

    s=sock.getinband 'we get entire CMD buffer contents into the s
    x=instr(1,s,chr(sock.endchar))
    while x<>0
        s2=left(s,x-1) 's2 now contains a single inband command
        s=right(s,len(s)-x) 'cut out this command

        'process inband command in the s2 as needed
        ...
        ...

        'any more inband commands to process now?
        x=instr(1,s,chr(sock.endchar))
    wend
end sub

```



For the above example to work well, the size of the CMD buffer must not exceed the capacity of string variable *s*. This way whatever is extracted from the CMD buffer will always fit in *s*. A slightly more complex processing is needed if the buffer is larger than the capacity of *s*.

Are CMD buffer overruns possible?

CMD buffer overruns are not possible. If the socket receives an inband command that cannot be saved into the CMD buffer in its entirety, then the socket will discard the whole command (your program won't be notified of this in any way). Therefore, you are guaranteed to always receive complete inband commands, or nothing at all.

Incomplete inband commands

Take a look at this datastream:

Data@!comma@!command2&Moredata

What we have here is an inband command that is incomplete- a new inband command starts before the previous one ends. Such incomplete commands are discarded and not recorded into the CMD buffer.

Sending Inband Replies

How to generate inband reply correctly

Inband replies are sent from the RPL buffer. Unlike the process of sending "regular" TX data that requires you to use [sock.setdata](#)^[353] and [sock.send](#)^[353] methods, the process of sending an inband reply only takes one step. You set and send (commit) the data with a single method- [sock.setsendinband](#)^[354].

The `sock.setsendinband` method puts the data into the RPL buffer and immediately commits it for sending. The socket does not add necessary encapsulation automatically: it is the responsibility of your application to add the escape character, some other character after the escape, and the end character.

Your inband reply will only be stored into the RPL buffer if the latter has enough space to store your entire message. If there is not enough free space then *nothing* will be stored. This is different from the TX buffer, for which *whatever can fit* is stored. You can check the free space in the RPL buffer by using the [sock.rplfree](#)^[350] property. Amount of unsent data in the RPL buffer can be checked through the [sock.rpllen](#)^[350] property.

You must not split your inband reply- it must be placed in the RPL buffer with a single invocation of `sock.setsendinband`. In the example below we reply back "OK" to each inband command we receive:

```
sub on_sock_inband
    dim s as string 'we will keep the data from the CMD buffer here
    dim s2 as string 'this will keep individual inband commands
    dim x as byte

    s=sock.getinband 'we get entire CMD buffer contents into the s
    x=instr(1,s,chr(sock.endchar))
    while x<>0
        s2=left(s,x-1)  's2 now contains a single inband command
        s=right(s,len(s)-x)  'cut out this command

        'reply back with OK
        sock.setsendinband(chr(sock.escchar)+" OK"+chr(sock.endchar))

        'any more inband commands to process now?
        x=instr(1,s,chr(sock.endchar))
    wend
end sub
```

And this would be an incorrect way- *do not* split inband replies!

```
...
    sock.setsendinband(chr(sock.escchar))      'WRONG!
    sock.setsendinband(" OK")                  'WRONG!
    sock.setsendinband(chr(sock.endchar))      'WRONG!
...

```



For the above example to work well, the size of the CMD buffer must not exceed the capacity of string variable `s`. This way whatever is extracted from the CMD buffer will always fit in `s`. A slightly more complex processing is needed if the buffer is larger than the capacity of `s`.



Notice how we have created a character after the escape character- by adding a space in front of our "OK" reply, like this: " OK". This will work fine as long as our escape character is not space!

Using HTTP

The `sock` object can function as a HTTP server. This means that when certain conditions are met, individual sockets will switch into the HTTP mode and output the data in a style, consistent with the HTTP server functionality.

Certain BASIC information about the HTTP server has already been provided in [Working with HTML](#)^[74] and [Embedding Code Within an HTML File](#)^[76].

When the socket is in the HTTP mode your program has no control over the received data (HTTP requests) and only *sometimes* has control over the

transmitted data (HTTP reply).

In the simplest case the file returned to the web browser is static- a "fixed" HTML page, a graphic, or some other file. Processing of such a static file requires no intervention from your program whatsoever. Just setup the socket(s) to be able to accept HTTP requests and the sock object will take care of the rest.

More often than not, however, you have to create a dynamic HTML page. Dynamic pages include fragments of BASIC code. When the sock object encounters such a fragment in the file being sent to the browser, it executes the code. This code, in turn, performs some action, for example, generates and sends some dynamic data to the browser or jumps to the other place in the HTML file.

The HTTP server built into the sock object understands two request types- GET and POST. Both can carry "HTTP variables" that the server will extract and pass to the BASIC code.

At the moment, the following file types are explicitly supported:

- HTML- can be static or include BASIC code.
- TXT- plain text, no BASIC code can be included.
- JPG and GIF- graphic files.

All files other than HTML files are static and are sent to the browser "as is". There is, however, a method of programmatic generation of such files -- see [URL Substitution](#)^[322].

All other file types are handled as binary files.



Currently, the socket object can only access first 65534 bytes of each HTML file, even if the actual file is larger! Make sure that all HTML files in your project are not larger than 65534 bytes. This is not to be confused with the size of HTTP output generated by the file. A very large output can be generated by a small HTML file (due to dynamic data generation)- and that is OK. What's important is that the size of each HTML file in your project does not exceed 65534 bytes.

65534 is actually the size limitation for the *compiled* HTML file. When compiling your project, the TIDE will separate the static portion of the file from the Tibbo Basic code fragments. Only the compiled file size matters.

HTTP-related Buffers

For the HTTP to work, you need to allocate some memory to the following buffers:

- RX buffer. This buffer will be receiving HTTP requests from the client (browser). Buffer allocation request is done through the [sock.rxbuffrq](#)^[350] method. It is possible to avoid spending memory on the RX buffer by [redirecting](#)^[308] this buffer to the TX buffer of the same socket. This will work because the web server operation is strictly sequential -- receive a request, then generate a reply. Requests and replies do not have to be stored concurrently, so one buffer is sufficient and you will save some memory!
- TX buffer. This buffer will be handling web server replies. If you enable redirection it will also receive HTTP requests. Buffer allocation request is done through the [sock.txbuffrq](#)^[362] method. Practical experience shows that allocating just one page for this buffer makes HTTP request handling somewhat slow. At

three or four pages, there is a significant performance improvement. Additional buffer pages do not lead to any dramatic improvements in performance.

- VAR buffer. Dynamic HTML pages include snippets of BASIC code and this code may need to know variable passed to the HTTP server using GET or POST methods. The VAR buffer stores those variables. Do the allocation with the [sock.varbuffrq](#)^[365] method. Buffer size of one page is usually OK. If the application is to handle large amounts of variable data, such as in the case of file uploads, you can improve the performance by allocating more pages.

Actual memory allocation is done through the [sys.buffalloc](#)^[217] method which applies to all buffers previously specified. Here is an example:

```
Dim f As Byte

...
'setup for other objects, sockets, etc.

'setup buffers of sockets 8-15 (will be used for HTTP)
For f=8 To 15
    sock.num=f
    sock.txbuffrq(1) 'you need this buffer for HTTP requests and replies
    sock.varbuffrq(1) 'you need this buffer to get HTTP variables
    sock.redirect(PL_REDIREC_SOCCK0 + sock.num) 'this will allow us to avoid
    wasting memory on the RX buffer
Next f

....
sys.buffalloc ' Performs actual memory allocation, as per previous
requests.
```

Memory allocation takes up to 100ms, so it is usually done just once, on boot, for all required buffers.

You may not always get the full amount of memory you have requested. Memory is not an infinite resource, and if you have already requested (and received) allocations for 95% of the memory for your platform, your next request will get up to 5% of memory, even if you requested for 10%.

There is a small overhead for each buffer. Meaning, not 100% of the memory allocated to a buffer is actually available for use. 16 bytes of each buffer are reserved for variables needed to administer this buffer, such as various pointers etc.

Thus, if we requested (and received) a buffer with 2 pages ($256 * 2 = 512$), we actually have 496 bytes in which to store data ($512 - 16$).



If you are *changing* the size of any buffer for a socket using `sys.buffalloc`, and this socket is not closed ([sock.stateSimple](#)^[358] is not `PL_SSTS_CLOSED`), the socket will be automatically closed. Whatever connection you had in progress will be discarded. The socket will not be closed if its buffer sizes remain unchanged.



Notice, that in most cases you will need to reserve more than one socket for HTTP. The HTTP server may need to service multiple requests from different computers at the same time. Even for a single computer and a single HTML page, more than one socket may be needed. For example, if your HTML page contains a picture, the browser will establish two parallel connections to the sock object- one to get the HTML page itself, another one- to get the picture. We recommend that you reserve 4-8 sockets for the HTTP. It is better to have less buffer memory for each HTTP sockets than to have fewer HTTP sockets!

Setting the Socket for HTTP

How to set the socket for HTTP

Apart from assigning some memory to the TX, RX, and VAR buffers, the following needs to be done to make the socket work in HTTP mode:

- The protocol must be TCP ([sock.protocol](#)^[345]= 1- PL_SOCK_PROTOCOL_TCP).
- The socket must be open for incoming connections (typically, from anybody: [sock.inconmode](#)^[338]= 3- PL_SOCK_INCONMODE_ANY_IP_ANY_PORT).
- Reconnects should not be enabled- this is counter-productive for HTTP ([sock.reconmode](#)^[345]= 0- PL_SOCK_RECONMODE_0). Reconnects are disabled by default so just leave it this way.
- Correct listening HTTP port must be set. Default HTTP port on all servers is 80 ([sock.httpportlist](#)^[335]="80").

In the previous topic, we have already explained that your system should reserve several HTTP sockets. Here is a possible initialization example:

```
dim f as byte

...
'setup for other sockets, etc.

'setup sockets 8-15 for HTTP
for f=8 to 15
    sock.num=f
    sock.protocol= PL_SOCK_PROTOCOL_TCP
    sock.inconmode= PL_SOCK_INCONMODE_ANY_IP_ANY_PORT
    sock.httpportlist="80"
next f
....
```

To make sure that the HTTP is working you can create and add to the project a simple static HTTP file. Call this file <index.html>- this is a default file that will be called if no specific file is requested by GET or POST. Here is a static file example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD W3 HTML//EN">
<HTML>
<BODY>
    HELLO WORLD!<br>
</BODY>
</HTML>
```

Launch the browser, type the IP-address of your device, for example: "http://192.168.1.95" and you will get the output "HELLO WORLD!".



Don't forget to give your device an IP address. For example, if you are working through a regular wired Ethernet, you should assign an IP address through the [net.ip](#)^[270] property.

Double-duty: non-HTTP and HTTP processing on the same socket

Your HTTP sockets don't have to be *exclusively* HTTP. You can have them behave differently depending on which listening port the TCP connection is being made to. Here is an example: supposing the setup of your device needs to be effected in two ways- via TELNET or via HTTP. Standard TELNET port is 23, standard HTTP port is 80. Setup your socket like this:

```
sock.num=f
sock.protocol= PL_SOCK_PROTOCOL_TCP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IP_ANY_PORT
sock.localportlist="23"
sock.httpportlist="80"
....
```

The socket will now be accepting connections both on port 23 and port 80. When connection is made to port 23, the socket will work as a regular data socket, as was described in previous sections. When connection is made to port 80, the socket will automatically switch into the HTTP mode!

There is a property- [sock.httpmode](#)^[334]- that tells you which mode the socket is in- regular data mode or HTTP. You can even "forcefully" switch any TCP connection into the HTTP mode by setting `sock.httpmode= 1- YES`. You cannot, however, switch this connection back to the data mode, it will remain in the HTTP mode until termination.

Socket Behavior in the HTTP Mode

When in the HTTP mode, the socket is behaving differently compared to the normal data mode.

Incoming connection rejection

As was explained in [Accepting Incoming Connections](#)^[279], if your device "decides" to reject an incoming TCP connection, it will send out a reset TCP packet. This way, the other host is instantly notified of the rejection. Rules are different for HTTP sockets:

- If there is an incoming TCP connection to the web server of the device (incoming HTTP connection request), and if your application has one or more sockets that are configured to accept this connection, and if all such sockets are already occupied, then the system will not reply to the requesting host *at all*.

- If there is an incoming HTTP connection request, and if your application has no HTTP sockets configured to accept this connection, then the system will still respond with a reset.

This behavior allows your application to get away with fewer HTTP sockets. Here is why. If all HTTP sockets are busy and your application sends out a reset, the browser will show a "connection reset" message. If, however, your device does not reply at all, the browser will wait, and resend its request later. Browsers are "patient" -- they will typically try several times before giving up. If any HTTP sockets are freed-up during this wait, the repeat request will be accepted and the browser will get its page. Therefore, very few HTTP sockets can handle a large number of page requests in a sequential manner and with few rejections.

Other differences

- All incoming data is still stored in the TX buffer (yes, TX buffer). This data, however, is not passed to your program but, instead, is interpreted as HTTP request. This HTTP request must be properly formatted. The `sock` object supports GET and POST commands.
- The RX buffer is [not used](#)^[315] at all and does not have to be allocated any memory.
- GET and POST commands can optionally contain "request variables". These are stored into the VAR buffer from which your program can read them out later.
- No [on_sock_data_arrival](#)^[342] event is generated when the HTTP request string is received into the RX buffer.
- Once entire request has been received the socket prepares and starts to output the reply. Your program has no control over this output until a BASIC code fragment is encountered in the HTTP file. The [on_sock_data_sent](#)^[342] event cannot be used as well.
- When code fragment is encountered in the HTTP file control is passed to it and then your program can perform desired action, i.e. generate some dynamic HTML content, etc. When this fragment is entered, the [sock.num](#)^[340] is automatically set to the correct socket number.
- Once HTTP reply has been sent to the client the socket will automatically close the connection, as is a normal socket behavior for HTTP. A special property-[sock.httpnorelease](#)^[335]- allows you to change this default behavior and leave the connection opened.

Including BASIC Code in HTTP Files

To create dynamic HTML pages, you include BASIC statements directly into the HTML file. A fragment of BASIC code is included within "<? ?>" encapsulation, as shown in the example below (for more info see [Working with HTML](#)^[74]):

```
<!DOCTYPE HTML public "-//W3C//DTD W3 HTML//EN">
<HTML>
<BODY>
    HELLO HTML WORLD!<br>
    <?
        'This is a BASIC code snippet. We can, for instance, send
        something to the serial port
        ser.num=0
        ser.setdata("HELLO SERIAL WORLD TOO!")
        ser.send
    ?>
</BODY>
</HTML>
```

Each time this HTML page will be requested by the browser, the "HELLO SERIAL WORLD TOO!" string will be sent out of the serial port.

Generating Dynamic HTML Pages

How to "print" dynamic HTML data

In most cases the BASIC code included into the HTML page is used to generate dynamic HTML content, i.e. to send some dynamically generated data to the browser. This is done in a way, similar to sending out data in a regular data mode: you first set the data you want to send with the [sock.setdata](#)^[353] method, then commit this data for sending using [sock.send](#)^[353] method.

Here is an example HTML page that checks the state of one of the inputs of the I/O object and reports this state on the HTML page:

```
<!DOCTYPE HTML public "-//W3C//DTD W3 HTML//EN">
<HTML>
<BODY>
    The state of line 0 is!:<br>
    <?
        'This is a BASIC code fragment. Get the state of the line.
        io.num=0
        if io.state= LOW then
            sock.setdata("LOW")
        else
            sock.setdata("HIGH")
        end if
        sock.send
    ?>
</BODY>
</HTML>
```



Notice how we did not have to use [sock.num](#)^[340]- it is set automatically when control is passed to BASIC!

Be careful not to get your data truncated!

The above example is seemingly correct and in this particular case will work fine- because there is not much static data preceding the point where we generate dynamic content and not much dynamic content is generated as well. Generally speaking, you cannot expect that the TX buffer will have enough space when you need to put some data into it! If you are not careful the dynamic data you want to generate may get truncated!

To avoid this situation always check if the necessary amount of free space is available before attempting to put it into the TX buffer. This statement is true for the normal data mode as well, not just for the HTTP processing- we have already touched on this subject in [Sending Data](#)^[304]. What is different for HTTP is that you cannot use the [on_sock_data_sent](#)^[342] event to "call you back" when the TX buffer frees up!

The only solution in case of HTTP is to wait, in a loop, for the desired amount of space to become available. Naturally, we don't want to be blocking the whole

device, so "polite" waiting shall include a [doevents](#)⁸² statement:

```
<?
    dim s as string(4)
?>

<!DOCTYPE HTML public "-//W3C//DTD W3 HTML//EN">
<HTML>
<BODY>
    The state of line 0 is!:!<br>
    <?
        'This is a BASIC code snippet. Get the state of the line.
        io.num=0
        if io.state= LOW then
            s="LOW"
        else
            s=HIGH
        end if

        'and now we wait till the TX buffer has enough free space
        while sock.txfree<len(s)
            doevents
        wend

        'OK, so necessary space is now available!
        sock.setdata(s)
        sock.send
    ?>
</BODY>
</HTML>
```

Now we have a bullet-proof dynamic content generation *and* non-blocking operation! As you get more experience with HTML, you will see that the doevents statement has to be used quite often.

Special case: doevents and concurrent request of the same file

With HTTP it is entirely possible that two computers (browsers) will request the same HTML file. If you have allocated more than one socket for HTTP it is also possible that both of those requests will be processed at the same time. Of course, the BASIC VM is a single-process system, so when it comes to processing dynamic part of the files, one of the requests will be first.

It is also entirely possible, that when executing the BASIC procedure from the first "instance" of the file, execution arrives at doevents and the second instance of the same file will start to process. After all, doevents allows other events to execute, so this can include activity in other sockets.

Now, if the same BASIC procedure was allowed to execute again this would cause recursion: an attempt to execute a portion of code while it is already executing! [Recursion is not allowed](#)⁶², so the VM will not re-enter the code immediately.

Instead, the second instance of HTML page will be "on hold" until the first instance finishes running the procedure in question. After that, the same procedure will be executed for the second instance. This behavior has been introduced in **V1.2** of TiOS. In the previous release, procedure execution for the second instance of the HTML page was simply skipped and not executed at all, which is wrong!

Once again, for this to occur the following conditions must be met: the HTML procedure in question must use doevents statement, and the same file (HTML page) must be requested simultaneously from several browsers.

Avoiding illegal characters

HTTP restricts the use of some characters- you cannot freely send any code from the ASCII table. When you use the `sock.setdata` and `sock.send`, you are "printing" data directly to the browser. It is your program's responsibility to avoid illegal characters and use "%xx" instead.

URL Substitution

HTML files can [include BASIC code](#)^[319] and thus be [dynamic](#)^[320]. Actual HTML contents received by the browser may partially be generated by your Tibbo BASIC application. Files of other types -- plain text, graphics, etc. -- cannot include executable code and are static in nature. These files are sent to the browser "as is".

So, what if you need to generate such a non-HTTP file dynamically? For example, what if you want to generate a BMP file? The answer is in using "URL substitution". The [sock.urlsubstitutes](#)^[364] property allows you to do this.

The property stored a list of comma-separated file names (with extensions). When the web server receives a request for a non-HTML file from the browser (say, "pix.bmp"), it first tries to find this file among the HTML and resource files of your project. Failing this, the web server looks at defined substitutions. If the requested file is not on the list, the web server returns the "404 error".

If the file is on the list of defined substitutions, the browser looks for the file with the same name but ".html" extension ("pix.html"). If there is no such file, the "404 error" is, again, the answer. If this HTML file exists the server outputs this file, but makes it look like it was a file of the original type (server processes "pix.html", browser gets "pix.bmp").

Since the actual behind-the-scenes output is done for the HTML file you can put your own code into that file and generate content dynamically. The following example shows how we generate the file "pix.bmp" dynamically. For this to work, we need to set `sock.urlsubstitutes="pix.bmp"` somewhere in the initialization section of the project. Here is what we have in "pix.html" file:

```
<?
  Dim x As Byte
  Dim s As String

  romfile.open("source.bmp")
  Do
    x=sock.txfree
    s=romfile.getdata(x)
    sock.setdata(s)
    sock.send
  Loop While Len(s)=x
?>
```

In this example we simply read and output the contents of another bmp file called "source.bmp". No value is added -- we could just access that graphical file directly. Your code, however, is free to do anything and substitution opens up the way to create your picture on the fly, or dynamically generate the content for other "static" files.

Working with HTTP Variables

HTML pages often pass "variables" to each other. For example, if you have an HTML form to fill on page "form.html" you may want to have the data input by the user on page "result.html".

The explanation of HTTP variables is divided into three sections:

- [Simple case](#)^[323] -- the amount of HTTP variable data does not exceed 255 bytes;
- [Complex case](#)^[324] -- the amount of HTTP variable data exceeds 255 bytes (can easily happen, especially with file uploads);
- [Details](#)^[325] on how the variable data actually looks to your application depending on the method used (HTTP GET or POST).

In most cases the client will send a fairly small amount of variable data, so it all will fit in 255 bytes. Use the [sock.httprqstring](#)^[336] R/O property, as shown in the following example. Say, you have a login to perform. The user enters his/her name and password on page "index.html". Actual login is processed on page "login.html". The data is passed between these two pages "invisibly", using the HTTP POST method.

Here is the file "index.html":

```
<html> <body>
    <form action="form_get.html" method="post"
enctype="multipart/form-data">
        username: <Input Type="text" name="user"><br>
        password: <Input Type="password" name="pwd"><br>
        <Input Type="submit" value="send">
    </form>
</body> </html>
```

And here is "login.html":

```
<html><body>
    <?
        dim s as string
        s=sock.httprqstring
        'actual processing here, s now contains the variable string
    ?>
</body></html>
```

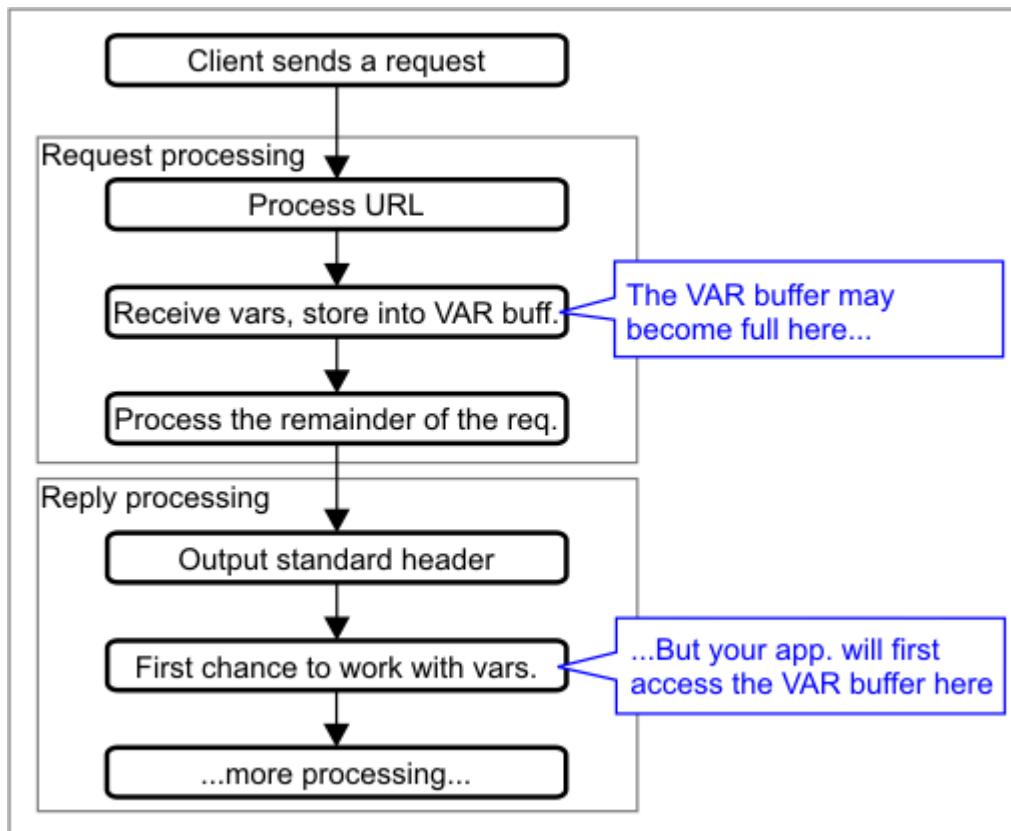
[Details on Variable Data](#)^[326] explains what exactly you get from the [sock.httprqstring](#)^[336] property.

The advantage of using the [sock.httprqstring](#) is that the variable data is always there, no matter how many times you read it. Be forewarned, though...



If you are using the [sock.httprqstring](#), and if the client sends more data than can fit in the VAR buffer, the execution of the HTTP request processing will be stalled indefinitely. See [Complex Case \(Large Amount of Variable Data\)](#)^[324] for a method of handling this correctly.

Let's take a bit about why the socket may get stuck. The following diagram details the flow of HTTP request and response processing:

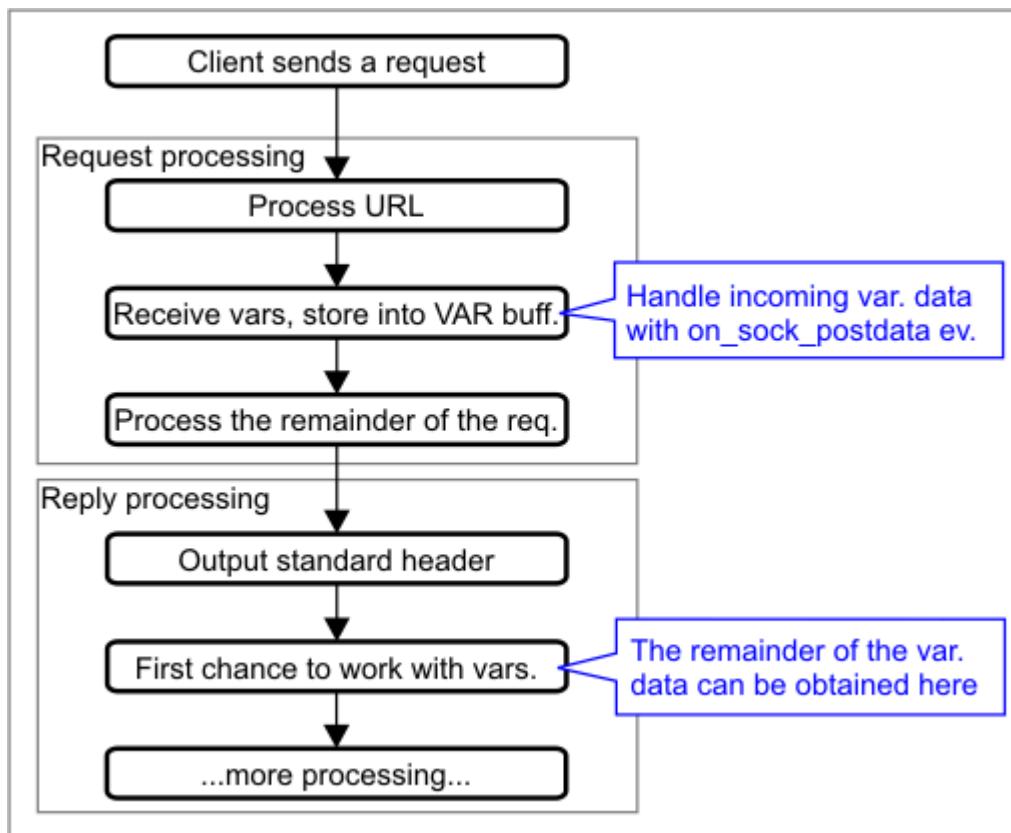


As you can see from the diagram, the socket will automatically extract the variable data and store it into the VAR buffer. The code example above will access the buffer during the reply phase -- from within the "login.html" page. Problem is, processing may never get that far. If the VAR buffer becomes full in the request processing phase, the socket will simply keep waiting, and the reply phase will never start! This is true no matter what HTTP method you use -- GET or POST. [Read on](#)^[324] and we will tell you how to avoid this.

To avoid a problem situation described in the [previous](#)^[323] topic use a more complex, but very reliable way of handling large HTTP variable data.

The [on_sock_postdata](#)^[344] event is generated when there is data in the VAR buffer. In this regard, the event is similar in nature to the [on_sock_data_arrival](#)^[342] event which is generated when the RX buffer of the socket has data. Unlike the `on_sock_data_arrival` event, the `on_sock_postdata` will first be generated only after the VAR buffer becomes full. That is, you won't be bothered by this event unless the HTTP request processing simply can't continue without it.

OK, so now you have a chance to access and process the HTTP variable data as it arrives and before the reply phase even starts. The [sock.gethttprqstring](#)^[333] method, unlike the [sock.httprqstring](#)^[336] R/O property, actually removes the data from the VAR buffer, thus freeing up the buffer space. Diagram below illustrates the process:



Here is a modified login example:

In the event handler for the `on_sock_postdata` event we extract available HTTP variable data and process it. The event will be called as many times as necessary. For example, we may save the data into a file:

```

Sub On_sock_postdata()
    fd.setdata(sock.gethttprqstring(255)) 'very simplified, but illustrates
    the point
End Sub
  
```

Then, in the HTML file. It is necessary to remember that a portion of the HTTP variable data may still be unhandled by the time you get here:

```

<html><body>
<?
    While sock.httprqstring<>"" 'extract the remaining part of the
variable data
        fd.setdata(sock.gethttprqstring(255))
    Wend
?>
</body></html>
  
```

It is unlikely that user login will require such careful handling. User name and password will comfortably fit in 255 bytes, unless your users are paranoid and have humongous passwords. Still, there are many situations when you need to send

large variable data. For example, HTTP POST methods are routinely used to upload files to the web server and your device will be able to handle this, too.

So far we haven't touched on the subject of actual data that you will read using `sock.httprqstring`^[338] and `sock.gethttprqstring`^[339]. In other words, we haven't discussed the contents of the VAR buffer. It is time to clear this.

We have already explained that the above R/O property and method will return the same data, with the only difference that the `sock.httprqstring` will not actually remove the data from the VAR buffer and will not be able to access past the first 255 bytes of such data.

Data format in the VAR buffer depends on the method used.

VAR buffer contents -- HTTP GET method

Supposing the client has sent the following request:

```
GET /form_get.html?user=CHUCKY&pwd=THEEVILDOLL HTTP/1.1
Host: 127.0.0.1:1000
User-Agent: Mozilla/5.0 (and so on)
```

The VAR buffer will get everything past the URL and until the first CR/LF. The part is highlighted in blue. There are two variables -- user and pwd. They are equal to "CHUCKY" and "THEEVILDOLL". Your application can just ignore the "HTTP/1.1" part.

VAR buffer contents -- HTTP POST method

Here is a sample client request:

```
POST /form_get.html HTTP/1.1
Host: 127.0.0.1:1000
User-Agent: Mozilla/5.0
(lots of stuff skipped)
Content-Type: multipart/form-data; boundary=-----
21724139663430
Content-Length: 257

-----21724139663430
Content-Disposition: form-data; name="user"

CHUCKY
-----21724139663430
Content-Disposition: form-data; name="pwd"

THEEVILDOLL
-----21724139663430--
```

You will get everything shown in **blue**.

Properties, Methods, and Events

This section provides an alphabetical list of all properties, methods, and events of the `sock` object.

.Acceptbcast Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) specifies whether the socket will accept incoming broadcast UDP datagrams.
Type:	Enum (yes_no, byte)
Value Range:	0- NO (default): will not accept broadcasts. 1- YES: will accept broadcasts.
See Also:	---

Details

This property is irrelevant for TCP communications ([sock.protocol](#)^[345] =PL_SOCKET_PROTOCOL_TCP).

.Allowedinterfaces Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) defines the list of network interfaces on which this socket will accept incoming connections.
Type:	String
Value Range:	Platform-specific. See the list of supported interfaces in the "Platform-dependent Programming Information" topic found inside the documentation for your platform.
See Also:	Accepting Incoming Connections ^[279]

Details

Each network interface has a name. For example, the Ethernet interface goes by the name "NET" (because it is handled by the [net](#)^[267] object). The Wi-Fi interface is called "WLN" (see [wln](#)^[497] object).

The list of allowed interfaces is comma-delimited, i.e. "WLN,NET". Note that reading back the value of this property will return the same list, but not necessarily in the same order. For example, the application may write "WLN,NET" into this property, yet read "NET,WLN" back. Unsupported interface names will be dropped from the list automatically.

The socket will not accept a connection on the interface which is not on the `sock`.

allowedinterfaces list, even if all other connection parameters such as protocol, port, etc. are correct.

This property is not available on the [EM202/200 \(-EV\), DS202](#)^[134] platform.

.Bcast R/O Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) reports whether the current or most recently received UDP datagram was a broadcast one.
Type:	Enum (no_yes, byte)
Value Range:	0- NO (default): the UDP datagram is not a broadcast one. 1- YES: the UDP datagram is a broadcast one.
See Also:	sock.remotemac ^[348] , sock.remoteip ^[347] , sock.remoteport ^[348]

Details

When the [on_sock_data_arrival](#)^[342] event handler is entered, the sock.bcast will contain the broadcast status for the current datagram being processed. Outside of the on_sock_data_arrival event handler, the property will return the broadcast status of the most recent datagram received by the socket.

.Close Method

Function:	For the selected socket (selection is made through sock.num ^[340]) causes the socket to close the connection with the other host.
Syntax:	sock.close
Returns:	---
See Also:	

Details

For established TCP connections this will be a "graceful disconnect", if the TCP connection was in the "connection opening" or "connection closing" state this will be a reset (just like when the [sock.reset](#)^[348] method is used). If connection was in the ARP phase or the transport protocol was UDP ([sock.protocol](#)^[345]= 0- 0- PL_SOCK_PROTOCOL_UDP) the connection will be discarded (just like when the [sock.discard](#)^[331] method is used). Method invocation will have NO effect if connection was closed at the time when the method was called ([sock.state](#)^[355] in one of PL_SST_CLOSED states).

This method will be ignored when called from within an HTML page. HTML sockets are handled automatically and your application is not at freedom to close HTML

sockets arbitrarily.

.Cmdbuffrq Method

Function:	For the selected socket (selection is made through sock.num ^[340]) pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the CMD buffer of the socket.
Syntax:	sock.cmdbuffrq(numpages as byte) as byte
Returns:	Actual number of pages that can be allocated (byte)
See Also:	sock.rplbuffrq ^[349]

Part	Description
numpages	Requested numbers of buffer pages to allocate.

Details

The CMD buffer is the buffer that accumulates incoming inband commands (messages). This method returns actual number of pages that can be allocated. Actual allocation happens when the sys.buffalloc method is used. The socket is unable to receive inband commands if its CMD buffer has 0 capacity. Unlike for TX or RX buffers there is no property to read out actual CMD buffer capacity in bytes. This capacity can be calculated as num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the [sock.cmdbuffrq](#)^[329] method. "-16" is because 16 bytes are needed for internal buffer variables.

Buffer allocation will not work if the socket port to which this buffer belongs is not idle ([sock.statesimple](#)^[358] is not at 0- PL_SSTS_CLOSED) at the time when sys.buffalloc executes. You can only change buffer sizes of sockets that are idle.

The CMD buffer is only required when inband commands are enabled ([sock.inbandcommands](#)^[337]= 1-YES).

.Cmdlen R/O Property

Function:	For the selected socket (selection is made through sock.num ^[340]) returns the length of data (in bytes) waiting to be processed in the CMD buffer.
Type:	Word
Value Range:	Default= 0 (0 bytes)
See Also:	sock.rpllen ^[350] , sock.inbandcommands ^[337]

Details

The CMD buffer accumulates incoming inband commands (messages) and may contain more than one such command. Use [sock.getinband](#)^[334] method to extract the data from the CMD buffer.

.Connect Method

Function:	For the selected socket (selection is made through sock.num ^[340]) causes the socket to attempt to connect to the target host.
Syntax:	sock.connect
Returns:	---
See Also:	

Details

The target is specified by the [sock.targetport](#)^[360] and [sock.targetip](#)^[359] (unless, for UDP, the socket is to broadcast the data- see the [sock.targetbroadcast](#)^[359] property). Outgoing connection will be attempted through the network interface defined by the [sock.targetinterface](#)^[359] property (**not supported** by the [EM202/200 \(-EV\)](#), [DS202](#)^[134] platform).

Method invocation will have effect only if connection was closed at the time when the method was called ([sock.state](#)^[355] in one of PL_SST_CLOSED states).

.Connectiontout Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) sets/returns connection timeout threshold for the socket in half-second increments.
Type:	Word
Value Range:	0-65535 (0-32767.5 seconds, 0 means "no timeout"), default = 0 (no timeout)
See Also:	Closing Connections ^[290]

Details

When no data is exchanged across the connection for `sock.connectiontout/2` number of seconds this connection is aborted (in the same way as if the [sock.reset](#)^[348] method was used). Connection timeout of 0 means "no timeout".

Actual time elapsed since the last data exchange across the socket can be obtained through the [sock.toutcounter](#)^[360] R/O property.

.Currentinterface R/O Property

Function:	For the currently selected socket (selection is made through sock.num) returns the network interface this socket is currently communicating through.
Type:	Enum (pl_sock_interfaces, byte)
Value Range:	Platform-specific. See the list of pl_sock_interfaces constants in the platform specifications.
See Also:	Checking Connection Status

Details

The value of this property is only valid when the socket is not idle, i.e. `sock.statesimple <> 0 - PL_SSTS_CLOSED`.

This property is not available on the [EM202/200 \(-EV\), DS202](#) platform.

.Discard Method

Function:	For the selected socket (selection is made through sock.num) causes the socket to discard the connection with the other host.
Syntax:	sock.discard
Returns:	---
See Also:	sock.close , sock.reset

Details

Discarding the connection means simply forgetting about it without notifying the other side of the connection in any way.

This method will be ignored when called from within an HTML page. HTML sockets are handled automatically and your application is not at freedom to discard HTML sockets arbitrarily.

.Endchar Property

Function:	For the currently selected socket (selection is made through sock.num) specifies the ASCII code of the character that will end inband command (message).
Type:	Byte
Value Range:	0-255, default = 13 (CR)
See Also:	sock.escchar

Details

Each inband message has to end with the end character, which will mark a return to the "regular" data stream of the TCP connection.

This property is irrelevant when inband commands are disabled ([sock.inbandcommands](#)^[337]= 0- NO). The program won't be able to change the value of this property when the socket is not idle ([sock.statesimple](#)^[358]<> 0-PL_SSTS_CLOSED).

.Escchar Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) specifies the ASCII code of the character that will be used as an escape character for inband commands (messages).
Type:	Byte
Value Range:	0-255, default = 255
See Also:	sock.endchar ^[337]

Details

Each inband message starts with "EC OC", where "EC" is the escape character defined by the [sock.escchar](#) property and "OC" is any character other than "EC". With inband commands enabled, data characters with code matching that of the escape character is transmitted as "EC EC" (that is, two identical characters are needed to transmit a single data character with code matching that of escape character).

This property is irrelevant when inband commands are disabled ([sock.inbandcommands](#)^[337]= 0- NO). The program won't be able to change the value of this property when the socket is not idle ([sock.statesimple](#)^[358]<> 0-PL_SSTS_CLOSED).

.Event R/O Property (Obsolete)

This property is no longer available. Instead, the [on_sock_event](#)^[343] property has a newstate argument that carries the state of the socket at the time of event generation.

.Eventsimple R/O Property (Obsolete)

This property is no longer available. Instead, the [on_sock_event](#)^[343] property has a newstatesimple argument that carries the simplified state of the socket at the time of event generation.

.Getdata Method

Function:	For the selected socket (selection is made through sock.num) returns the string that contains the data extracted from the RX buffer.
Syntax:	ser.getdata(maxinplen as word) as string
Returns:	String containing data extracted from the RX buffer
See Also:	

Part	Description
maxinplen	Maximum amount of data to return (word).

Details

Extracted data is permanently deleted from the buffer. Length of extracted data is limited by one of the three factors (whichever is smaller): amount of data in the RX buffer itself, capacity of the "receiving" string variable, and the limit set by the maxinplen argument.

Additionally, if this socket uses UDP transport protocol ([sock.protocol](#)= 1-PL_SOCKET_PROTOCOL_TCP) the length of data that will be extracted is limited to the UDP datagram being processed. Additional conditions apply to UDP datagram processing; see [on_sock_data_arrival](#) event and [sock.nextpacket](#) method.

.Gethttprqstring Method

Function:	For the selected socket (selection is made through sock.num) extracts up to 255 bytes of the HTTP request string from the VAR buffer.
Syntax:	ser.gethttprqstring(maxinplen as word) as string
Returns:	String containing data extracted from the VAR buffer.
See Also:	Working with HTTP Variables , Sock.httprqstring

Part	Description
maxinplen	Maximum amount of data to return (word).

Details

Extracted data is permanently deleted from the VAR buffer. VAR buffer contents are explained in [Details on Variable Data](#).

Length of extracted data is limited by one of the three factors (whichever is smaller): amount of data in the buffer itself, capacity of the "receiving" string variable, and the limit set by the maxinplen argument.

This method is only relevant when the socket is in the HTTP mode ([sock.httpmode](#)^[334]= 1- YES). Use it from within an HTML page or [on_sock_postdata](#)^[344] event handler.

.Getinband Method

Function:	For the selected socket (selection is made through sock.num ^[340]) returns the string that contains the data extracted from the CMD buffer.
Syntax:	sock.getinband as string
Returns:	String containing data from the CMD buffer
See Also:	

Part	Description
Details	

The CMD buffer is the buffer that accumulates inband commands. Extracted data is permanently deleted from the CMD buffer. Length of extracted data is limited by one of the two factors (whichever is smaller): amount of data in the CMD buffer itself, and the capacity of the "receiving" buffer variable. Several inband commands may be waiting in the CMD buffer. Each command will always be complete, i.e. there will be no situation when you will extract a portion of the command because the end of this command hasn't arrived yet. Inband commands stored in the CMD buffer will have escape character (see [sock.escchar](#)^[332] property) and the character after the escape character already cut off, but the end character (see [sock.endchar](#)^[331] property) will still be present. Therefore, your application can separate inband command from each other by finding the end characters.

.Httpmode Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) specifies whether this socket is in the HTTP mode.
Type:	Enum (no_yes, byte)
Value Range:	0- NO (default): "regular" TCP connection. 1- YES: - HTTP connection.
See Also:	---

Part	Description
Details	

This property is irrelevant when the [sock.protocol](#)^[345]= PL_SOCK_PROTOCOL_UDP

(UDP). If you do not set this property directly, its value will be:

0- NO: for all outgoing connections (active opens) of the socket.

0- NO: for incoming connections received on one of the ports from the [sock.localportlist](#)^[339] list.

1- YES: for incoming connections received on one of the ports from the [sock.httpportlist](#)^[335] list.

You can manually switch any TCP connection at any time after it has been established from "regular" to HTTP by setting `sock.httpmode= 1`. However, this operation is "sticky"- once you have converted the TCP connection into the HTTP mode you cannot convert it back into the regular mode- trying to set `sock.httpmode=0` won't have any effect- the TCP connection will remain in the HTTP mode until this connection is closed.

.Httpnoclose Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) sets/returns whether TCP HTTP connection will be kept opened after the HTTP request has been processed and the HTML page has been sent out
Type:	Enum (no_yes, byte)
Value Range:	0- NO (default): connection will be closed (standard HTML server behavior) 1- YES: will be kept opened, special string will be used as a separator.
See Also:	---

Details

Normally, the end of HTML output is indicated by closing the TCP connection. When this property is set to 1- YES, connection is not closed at the end of HTML output. As a substitute, the end of HTML page output is marked by the following string: "<tibbo_linkserver_http_proxy_end_of_output>".

.Httpportlist Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) sets/returns the list of listening ports on any of which this socket will accept an incoming HTTP connection.
Type:	String
Value Range:	0-32 characters, default = ""
See Also:	sock.localport ^[338] , sock.httpmode ^[334]

Details

This property is only relevant when incoming connections are allowed by the [sock.inconmode](#)^[338] property and the the [sock.protocol](#)^[345]= 1-PL_SOCK_PROTOCOL_TCP (HTML output cannot be done through the UDP).

This property is of string type and the list of ports is a comma-separated string, i.e. "80,81" (to accept HTTP connections on either port 80 or 81). Max string length for this property is 32 bytes.

Notice, that there is also the [sock.localportlist](#)^[339] property that defines a list of listening ports for UDP and non-HTTP TCP connections. When a particular port is listed both under the sock.localportlist and the sock.httpportlist, and the protocol for this socket is TCP then sock.httpportlist has precedence (incoming TCP connection on the port in question will be interpreted as HTTP).

For example, if the `sock.httpportlist= "80,81"`, the `sock.localportlist="3000,80"`, the `sock.protocol= 1- PL_SOCK_PROTOCOL_TCP`, and there is an incoming TCP connection request on port 80 then this connection will be interpreted as HTTP one.

.Httprqstring R/O Property

Function:	For the selected socket (selection is made through sock.num ^[340]) returns up to 255 bytes of the HTTP request string stored in the VAR buffer.
Type:	String
Value Range:	Default= ""
See Also:	Sock.varbuffrq ^[365]

Details

The `sock.httprqstring` is a property; it can be invoked several times and will return the same data (when this property is used the data is not deleted from the VAR buffer). VAR buffer contents are explained in [Details on Variable Data](#)^[326].

This property is only relevant when the socket is in the HTTP mode ([sock.httpmode](#)^[334]= 1- YES). Use it from within an HTML page or [on_sock_postdata](#)^[341] event handler. Maximum length of data that can be obtained through this property is 255 bytes, since this is the maximum possible capacity of a string variable that will accept the value of the `sock.httprqstring`.

HTTP requests can be much longer than 255 bytes and can even include entire files being uploaded from the client to your device. Rely on the `on_sock_postdata` event and the [sock.gethttprqstring](#)^[333] method to handle large amounts of HTTP variable data correctly.



If you are using the `sock.httprqstring`, and if the client sends more data than can fit in the VAR buffer, the execution of the HTTP request will be stalled indefinitely. To avoid this, rely on the `on_sock_postdata` event and the `sock.gethttprqstring` method, as explained in [Complex Case \(Large Amount of Variable Data\)](#)^[324].

.Inbandcommands Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) specifies whether inband command passing is allowed.
Type:	Enum (no_yes, byte)
Value Range:	0- NO (default): inband commands are not allowed. 1- YES: inband commands are allowed.
See Also:	---

Details

Inband commands are messages passed within the TCP data stream. Each message has to be formatted in a specific way- see the [sock.escchar](#)^[332] and [sock.endchar](#)^[331] properties.

Inband commands are not possible for UDP communications so this setting is irrelevant when the [sock.protocol](#)^[345]= 1- PL_SOCKET_PROTOCOL_UDP. Inband messaging will work even when redirection (buffer shorting) is enabled for the socket (see the [sock.redir](#)^[346] method). The program won't be able to change the value of this property when the socket is not idle ([sock.statesimple](#)^[358]<> 0- PL_SSTS_CLOSED).

.Inconenabledmaster Property

Function:	A master switch that globally disables incoming connection acceptance on all sockets, regardless of each socket's individual setup.
Type:	Enum (no_yes, byte)
Value Range:	0- NO: No socket will be allowed to accept an incoming connection. 1- YES (default): Incoming connections are globally enabled. Individual socket's behavior and whether it will accept or reject a particular incoming connection depends on the setup of this socket.
See Also:	sock.inconmode ^[338] , sock.localportlist ^[339] , sock.httpportlist ^[335]

Details

This property can be used to temporarily disable incoming connection acceptance on all sockets without changing individual setup of each socket.

.Inconmode Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) specifies whether incoming connections (passive opens) will be accepted and, if yes, from which sources.
Type:	Enum (pl_sock_inconmode, byte)
Value Range:	0- PL_SOCK_INCONMODE_NONE (default): The socket does not accept any incoming connections. 1- PL_SOCK_INCONMODE_SPECIFIC_IPPORT: The socket will only accept an incoming connection from specific IP (matching sock.targetip ^[359]) and specific port (matching sock.targetport ^[360]) 2- PL_SOCK_INCONMODE_SPECIFIC_IP_ANY_PORT: The socket will only accept an incoming connection from specific IP (matching sock.targetip), but any port. 3- PL_SOCK_INCONMODE_ANY_IP_ANY_PORT: The socket will accept an incoming connection from any IP and any port.
See Also:	sock.reconmode ^[345] , sock.localportlist ^[339] , sock.httpportlist ^[335]

Details

.Localport R/O Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) returns current local port of the socket.
Type:	Word
Value Range:	0-65535, default= 0
See Also:	---

Details

Your application cannot set the local port directly. Instead, a list of ports on which the socket is allowed to accept an incoming connection (passive open) is supplied via the [sock.localportlist](#)^[339] and [sock.httpportlist](#)^[335] properties.

An incoming connection is accepted on any port from those two lists. The [sock.localport](#) property reflects current or the most recent local port on which connection was accepted.

.Localportlist Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) sets/returns the list of listening ports on any of which this socket will accept an incoming UDP or TCP connection.
Type:	String
Value Range:	0-32 characters, default = ""
See Also:	sock.localport ^[338]

Details

The socket will only accept an incoming connection when such connections are allowed by the [sock.inconmode](#)^[338] property. Whether the socket will accept UDP or TCP connections is defined by the [sock.protocol](#)^[345] property. Additionally, the [sock.allowedinterfaces](#)^[327] property (**not supported** by the [EM202/200 \(-EV\)](#), [DS202](#)^[134] platform) defines network interfaces on which the socket will accept an incoming connection.

This property is of string type and the list of ports is a comma-separated string, i.e. "1001,3000" (to accept connections on either port 1001 or 3000). Max string length for this property is 32 bytes. Notice, that there is also a [sock.httpportlist](#)^[335] property that defines a list of listening ports for HTTP TCP connections.

.Newtxlen R/O Property

Function:	For the selected socket (selection is made through sock.num ^[340]) returns the amount of uncommitted TX data in bytes.
Type:	Word
Value Range:	0-65535, default = 0 (0 bytes)
See Also:	---

Details

Uncommitted data is the one that was added to the TX buffer with the [sock.setdata](#)^[353] method but not yet committed using the [sock.send](#)^[353] method.

.Nextpacket Method

Function:	For the selected socket (selection is made through sock.num ^[340]) in the UDP mode (sock.protocol ^[345] = 0-PL_SOCKET_PROTOCOL_UDP) closes processing of current UDP datagram and moves to the next datagram.
------------------	---

Syntax: `sock.nextpacket`

Returns: ---

See Also: ---

Details

For UDP, the [sock.getdata](#)^[333] method only extracts the data from a single UDP datagram even if several datagrams are stored in the RX buffer. When incoming UDP datagram processing is based on the [on_sock_data_arrival](#)^[342] event the use of the `sock.nextpacket` method is not required since each invocation of the `on_sock_data_arrival` event handler "moves" processing to the next UDP datagram.

The method is useful when it is necessary to move to the next datagram without re-entering `on_sock_data_arrival` event handler. Therefore, `sock.nextpacket` is only necessary when the application needs to process several incoming UDP packets at once and within a single event handler.

.Notifysent Method

Function: Using this method for the selected socket (selection is made through `sock.num`) will cause the [on_sock_data_sent](#)^[342] event to be generated when the amount of committed data in the TX buffer is found to be below "threshold" number of bytes.

Syntax: `notifysent(threshold as word)`

Returns: ---

See Also:

Part	Description
threshold	Amount of bytes in the TX buffer below which the event it so be generated.

Details

Only one `on_sock_data_sent` event will be generated each time after the `sock.notifysent` is invoked. This method, together with the `on_sock_data_sent` event provides a way to handle data sending asynchronously.

Just like with [sock.txfree](#)^[363], the trigger you set won't take into account any uncommitted data in the TX buffer.

.Num Property

Function: Sets/returns the number of currently selected socket.

Type: Byte

Value Range: The value of this property won't exceed [sock.numofsock](#)^[341]-1 (even if you attempt to set higher value). **Default=** 0 (socket #0 selected).

See Also: ---

Details

Sockets are enumerated from 0. Most other properties and methods of this object relate to the socket selected through this property. Note that socket-related events such as [on_sock_data_arrival](#)^[342] change currently selected socket!

.Numofsock R/O Property

Function: Returns total number of sockets available on the current platform.

Type: Byte

Value Range: platform-dependent

See Also: [sock.num](#)^[340]

Details

.Outport Property

Function: For the currently selected socket (selection is made through [sock.num](#)^[340]) sets/returns the number of the port that will be used by the socket to establish outgoing connections.

Type: Word

Value Range: 0-65535, **default=** 0

See Also: ---

Details

If this property is set to 0 then the socket will use "automatic" port numbers: for the first connection since the powerup the port number will be selected randomly, for all subsequent outgoing connections the port number will increase by one. Actual local port of a connection can be queried through the [sock.localport](#)^[338] read-only property.

If this property is not at zero then the port it specifies will be used for all outgoing

connections from this socket.

On_sock_data_arrival Event

Function:	Generated when at least one data byte is present in the RX buffer of the socket (i.e. for this socket the sock.rxlen ^[352] >0).
Declaration:	on_sock_data_arrival
See Also:	sock.nextudptagram ^[339]

Details

When the event handler for this event is entered the [sock.num](#)^[340] property is automatically switched to the socket for which this event was generated. Another [on_sock_data_arrival](#)^[342] event on a particular socket is never generated until the previous one is processed.

Use the [sock.getdata](#)^[333] method to extract the data from the RX buffer.

For TCP protocol ([sock.protocol](#)^[345]= 1- PL_SOCK_PROTOCOL_TCP), there is no separation into individual packets and you get all arriving data as a "stream". You don't have to process all data in the RX buffer at once. If you exit the [on_sock_data_arrival](#) event handler while there is still some unprocessed data in the RX buffer another [on_sock_data_arrival](#) event will be generated immediately.

For UDP protocol (sock.protocol= 0- PL_SOCK_PROTOCOL_UDP), the RX buffer preserves datagram boundaries. Each time you enter the [on_sock_data_arrival](#) event handler you get to process next UDP datagram. If you do not process entire datagram contents the unread portion of the datagram is discarded once you exit the event handler.

This event is not generated for a particular socket when buffer redirection is set for this socket through the [sock.redir](#)^[346] method.

On_sock_data_sent Event

Function:	Generated after the total amount of <i>committed</i> data in the TX buffer of the socket (sock.txlen) is found to be less than the threshold that was preset through the sock.notifysent ^[340] method.
Declaration:	on_sock_data_sent
See Also:	---

Details

To cause generation of the [on_sock_data_sent](#) event, the application needs to use [sock.notifysent](#) each time. When the event handler is entered the [sock.num](#)^[340] is automatically switched to the socket on which this event was generated.

Please, remember that uncommitted data in the TX buffer is not taken into account for on_sock_data_sent event generation.

On_sock_event Event

Function: Notifies your program that the socket state has changed.

Declaration: **on_sock_event(newstate as pl_sock_state,
newstatesimple as pl_sock_state_simple)**

See Also: ---

Part	Description
newstate	"Detailed" state of the socket at the time of event generation.
newstatesimple	"Simplified" state of the socket at the time of event generation.

Details

The newstate and newstatesimple arguments carry the state as it was at the moment of event generation. This is different from [sock.state](#)^[355] and [sock.statesimple](#)^[358] R/O properties that return *current* socket state). The newstate argument uses the same set of constants as the sock.state. The newstatesimple argument uses the same set of constants as the sock.statesimple. See [sock.state](#)^[355] and [sock.statesimple](#)^[358] for detailed description of reported socket states.

Newstate and Newstatesimple arguments of the on_sock_event have been introduced in the Tibbo Basic release 2.0. They replace [sock.event](#)^[332] and [sock.eventsimple](#)^[332] R/O properties which are no longer available.

On_sock_inband Event

Function: At least one data byte is present in the CMD buffer ([sock.cmdlen](#)^[329]>0).

Declaration: **on_sock_inband**

See Also: ---

Details

Use the [sock.getinband](#)^[334] method to extract the data from the CMD buffer. Another on_inband_command event on a particular socket is never generated until the previous one is processed. When the event handler is entered the sock.num is automatically switched to the socket on which this event was generated.

On_sock_overrun Event

Function: Data overrun has occurred in the RX buffer of the socket.

Declaration: **on_sock_overrun**

See Also: ---

Details

Normally, overruns can only happen for UDP communications as UDP has no "data flow control" and, hence, data overruns are normal. Another on_sock_overrun event on a particular socket is never generated until the previous one is processed. When the event handler for this event is entered the [sock.num](#)^[340] is automatically switched to the socket on which this event was generated.

On_sock_postdata

Function: Generated when at least one data byte is present in the VAR buffer of the socket, but only after the VAR buffer has become full at least once in the cause of the current HTTP request processing.

Declaration: **on_sock_postdata**

See Also: [Sock.varbuffrq](#)^[365]

Details

HTTP requests can contain large amount of HTTP variable data, which is stored into the VAR buffer. The amount of such data can exceed the VAR buffer capacity. If this is not handled properly, the HTTP request execution may stall indefinitely -- see [Working with HTTP Variables](#)^[323].

After the socket accepts an HTTP connection, this event is not generated (for this particular connection) until the VAR buffer becomes full. Once this happened, the event is generated even if there is a single byte waiting to be processed in the buffer. Two same-socket on_sock_postdata events never wait in the queue -- the next event can only be generated after the previous one is processed.

When the event handler for this event is entered the [sock.num](#)^[340] property is automatically switched to the socket for which this event was generated.

Use the [sock.gethttprqstring](#)^[333] method or [sock.httprqstring](#)^[336] property to work with the VAR buffer's data.

On_sock_tcp_packet_arrival Event

Function: Notifies your program that the TCP packet of a certain size has arrived.

Syntax: **on_sock_tcp_packet_arrival(len as word)**

See Also: ["Split Packet" Mode of TCP Data Processing](#)^[306]

Part	Description
------	-------------

len Length of the new data in the RXed TCP packet.

Details

This event is only generated when [sock.splitcppackets](#)^[355]= 1- YES and [sock.inbandcommands](#)^[337]= 0- DISABLED. Notice that only new data, never transmitted before, is counted. If the packet is a retransmission then this event won't be generated. Also, if some part of packet's data is a retransmission and some part is new then only the length of the new data will be reported. This way your program can maintain correct relationship between data lengths reported by this event and actual data in the RX buffer.

The [on_sock_tcp_packet_arrival](#) event is always generated before the [on_sock_data_arrival](#)^[342] for any incoming TCP data (packet).

.Protocol Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) selects the transport protocol.
Type:	Enum (pl_sock_protocol, byte)
Value Range:	0- PL_SOCK_PROTOCOL_UDP (default): UDP transport protocol. 1- PL_SOCK_PROTOCOL_TCP: TCP transport protocol.
See Also:	---

Details

Notice, that there is no "HTTP" selection, as HTTP is not a transport protocol (TCP is the transport protocol required by the HTTP). You make the socket accept HTTP connections by specifying the list of HTTP listening ports using the [sock.httpportlist](#)^[335] property or using the [sock.httpmode](#)^[334] property.

The program won't be able to change the value of this property when the socket is not idle ([sock.statesimple](#)^[358]<> 0- PL_SSTS_CLOSED).

.Reconnmode Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) whether the socket accepts reconnects, and, if yes, from which sources.
Type:	Enum (pl_sock_reconnmode, byte)

Value Range:	<p>0- PL_SOCK_RECONMODE_0 (default): For UDP: Reconnects accepted only from the same IP as the one already engaged in the current connection with this socket, but any port; port switchover will not happen. TCP: reconnects are not accepted at all.</p> <p>1- PL_SOCK_RECONMODE_1: For UDP: Reconnects accepted from any IP, any port; port switchover will not happen. TCP: reconnects accepted only from the same IP and port as the ones already engaged in the current connection with this socket.</p> <p>2- PL_SOCK_RECONMODE_2: For UDP: Reconnects accepted only from the same IP as the one already engaged in the current connection with this socket, but any port; port switchover will happen. TCP: reconnects accepted only from the same IP as the one already engaged in the current connection with this socket, but any port.</p> <p>3- PL_SOCK_RECONMODE_3: For UDP: Reconnects accepted from any IP, any port; port switchover will happen. TCP: reconnects accepted from any IP, any port.</p>
---------------------	--

See Also:

Details

Reconnect situation is when a passive open and resulting connection replace, for the same socket, the connection that was already in progress. For UDP, this property additionally defines whether a "port switchover" will occur as a result of an incoming connection (passive open) or a reconnect. Port switchover is when the socket starts sending its outgoing UDP datagrams to the port from which the most recent UDP datagram was received, rather than the port specified by the [sock.targetport](#)^[360] property.

DO NOT enable reconnects on sockets that are used to handle HTML requests. This will interfere with HTML operation, as explained in the [Understanding TCP Reconnects](#)^[281] topic.

.Redir Method

Function: For the selected socket (selection is made through [sock.num](#)^[340]) redirects the data being RXed to the TX buffer of the same socket, different socket, or another object that supports compatible buffers.

Syntax: `sock.redir(redir as pl_redir) as pl_redir`

Returns: Returns 0- PL_REDIR_NONE if redirection failed or the same value as was passed in the redir argument.

See Also:

Part	Description
redir	Platform-specific. See the list of pl_redir constants in the platform specifications.

Details

Data redirection (sometimes referred to as "buffer shorting") allows to arrange efficient data exchange between ports, sockets, etc. in cases where no data alteration or parsing is necessary, while achieving maximum possible throughput is important.

The redir argument, as well as the value returned by this method are of "enum pl_redir" type. The pl_redir defines a set of platform inter-object constants that include all possible redirections for this platform. Specifying redir value of 0-PL_REDIR_NONE cancels redirection. When the redirection is enabled for a particular socket, the [on_sock_data_arrival](#)^[342] event is not generated for this port.

Once the RX buffer is redirected certain properties and methods related to the RX buffer will actually return the data for the TX buffer to which this RX buffer was redirected:

- [sock.rxbuffsize](#)^[351] will actually be returning the size of the TX buffer.
- [sock.rxclear](#)^[351] method will actually be clearing the TX buffer.
- [sock.rxlen](#)^[352] method will be showing the amount of data in the TX buffer.

.Remoteip R/O Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) returns the IP address of the host with which this socket had the most recent or currently has a connection.
Type:	String
Value Range:	Default = "0.0.0.0"
See Also:	sock.remotemac ^[348] , sock.remoteport ^[348] , sock.bcast ^[328]

Details

The application cannot directly change this property, it can only specify the target IP address for active opens through the [sock.targetip](#)^[359] property.

For UDP connections, when the [on_sock_data_arrival](#)^[342] event handler is entered, the sock.remoteip will contain the IP address of the sender of the current datagram being processed. Outside of the on_sock_data_arrival event handler, the property will return the source IP address of the most recent datagram received by the socket.

.Remotemac R/O Property

Function:	For the currently selected socket (selection is made through sock.num) returns the MAC address of the host with which this socket had the most recent or currently has a connection.
Type:	String
Value Range:	Default = "0.0.0.0.0.0"
See Also:	sock.remoteip , sock.remoteport , sock.bcast

Details

For UDP connections, when the [on_sock_data_arrival](#) event handler is entered, the sock.remotemac will contain the MAC address of the sender of the current UDP datagram being processed. Outside of the on_sock_data_arrival event handler, the property will return the source MAC address of the most recent datagram received by the socket.

.Remoteport R/O Property

Function:	For the currently selected socket (selection is made through sock.num) returns the port number of the host with which this socket had the most recent or currently has a connection.
Type:	Word
Value Range:	0-65535, default = 0
See Also:	sock.remotemac , sock.remoteip , sock.bcast

Details

The application cannot directly change this property, it can only specify the target port for active opens through the [sock.targetport](#) property.

For UDP connections, when the [on_sock_data_arrival](#) event handler is entered, the sock.remoteport will contain the port number of the sender of the current datagram being processed. Outside of the on_sock_data_arrival event handler, the property will return the source port of the most recent datagram received by the socket.

.Reset Method

Function:	For the selected socket (selection is made through sock.num) causes the socket to abort the connection with the other host.
------------------	--

Syntax:	sock.reset
Returns:	---
See Also:	sock.close ^[328]

Details

For TCP connections that were established, being opened, or being closed this will be a reset (RST will be sent to the other end of the connection). If connection was in the ARP phase or the transport protocol was UDP ([sock.protocol](#)^[345]= 0-PL_SOCK_PROTOCOL_UDP) the connection will be discarded (just like when the [sock.discard](#)^[331] method is used).

Method invocation will have NO effect if connection was closed at the time when the method was called ([sock.state](#)^[355] in one of PL_SST_CLOSED states).

This method will be ignored when called from within an HTML page. HTML sockets are handled automatically and your application is not at freedom to reset HTML sockets arbitrarily.

.Rplbuffrq Method

Function:	For the selected socket (selection is made through sock.num ^[340]) pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the RPL buffer of the socket.
Syntax:	sock.cmdbuffrq(numpages as byte) as byte
Returns:	Actual number of pages that can be allocated (Byte).
See Also:	sock.cmdbuffrq ^[329]

Part	Description
numpages	Requested numbers of buffer pages to allocate.

Details

The RPL buffer is the the buffer that stores outgoing inband replies (messages). Actual allocation happens when the [sys.buffalloc](#)^[217] method is used. The socket is unable to send inband replies if its RPL buffer has 0 capacity.

Unlike for TX or RX buffers there is no property to read out actual RPL buffer capacity in bytes. This capacity can be calculated as num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the [sock.rplbuffrq](#). "-16" is because 16 bytes are needed for internal buffer variables.

Buffer allocation will not work if the socket port to which this buffer belongs is not idle ([sock.statesimple](#)^[358] is not at 0- PL_SSTS_CLOSED) at the time when sys.buffalloc executes. You can only change buffer sizes of sockets that are idle.

The RPL buffer is only required when inband commands are enabled ([sock.inbandcommands](#)^[337]= 1- YES).

.Rplfree R/O Property

Function:	For the selected socket (selection is made through sock.num) returns the free space (in bytes) available in the RPL buffer
Type:	Word
Value Range:	0-65535, default = 0 (0 bytes)
See Also:	sock.cmdlen , sock.rpllen , sock.inbandcommands

Details

The RPL buffer is the buffer that keeps outgoing inband replies (messages). Your application adds inband replies to the RPL buffer with the [sock.setsendinband](#) method. Several inband replies may be waiting in the RPL buffer.

.Rpllen R/O Property

Function:	For the selected socket (selection is made through sock.num) returns the length of data (in bytes) waiting to be send out from the RPL buffer.
Type:	Word
Value Range:	
See Also:	sock.cmdlen , sock.rplfree , sock.inbandcommands

Details

Your application adds inband replies to the RPL buffer with the [sock.setsendinband](#) method. Several inband replies may be waiting in the RPL buffer.

.Rxbuffrq Method

Function:	For the selected socket (selection is made through sock.num) pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the RX buffer of the socket.
Syntax:	sock.rxbuffrq(numpages as byte) as byte
Returns:	Actual number of pages that can be allocated (Byte).
See Also:	sock.txbuffrq

Part	Description
numpages	Requested numbers of buffer pages to allocate.

Details

Returns actual number of pages that can be allocated. Actual allocation happens when the [sys.buffalloc](#)^[217] method is used. The socket is unable to RX data if its RX buffer has 0 capacity. Actual current buffer capacity can be checked through the [sock.rxbuffsize](#)^[351] which returns buffer capacity in bytes.

Relationship between the two is as follows: `sock.rxbuffsize=num_pages*256-16` (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the `sock.rxbuffrq`. "-16" is because 16 bytes are needed for internal buffer variables.

Buffer allocation will not work if the socket port to which this buffer belongs is not idle ([sock.statesimple](#)^[358] is not at 0- PL_SSTS_CLOSED) at the time when `sys.buffalloc` executes. You can only change buffer sizes of sockets that are idle.

.Rxbuffsize R/O Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) returns current RX buffer capacity in bytes.
Type:	Word
Value Range:	0-65535
See Also:	---

Details

Buffer capacity can be changed through the [sock.rxbuffrq](#)^[350]. The `sock.rxbuffrq` requests buffer size in 256-byte pages whereas this property returns buffer size in bytes.

Relationship between the two is as follows: `sock.rxbuffsize=num_pages*256-16` (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the `sock.rxbuffrq`. "-16" is because 16 bytes are needed for internal buffer variables. The socket cannot RX data when the RX buffer has zero capacity.

.Rxclear Method

Function:	For the selected socket (selection is made through sock.num ^[340]) clears (deletes all data from) the RX buffer.
Syntax:	sock.rxclear

Returns: ---**See Also:** ---**Details**

Invoking this method will have no effect when the socket is in the HTTP mode ([sock.httpmode](#)^[334]= 1- YES).

.Rxpacketlen R/O Property

Function: For the selected socket (selection is made through [sock.num](#)^[340]) returns the length (in bytes) of the UDP datagram being extracted from the RX buffer.

Type: Word

Value Range: 0-1536, **default**= 0

See Also: [sock.rxlen](#)^[352]

Details

This property is only relevant when the [sock.protocol](#)^[345]= 1-PL_SOCKET_PROTOCOL_TCP. Correct way of using this property is within the [on_sock_data_arrival](#)^[342] event or in conjunction with the [sock.nextpacket](#)^[339] method.

.Rxlen R/O Property

Function: For the selected socket (selection is made through [sock.num](#)^[340]) returns total number of bytes currently waiting in the RX buffer to be extracted and processed by your application.

Type: Word

Value Range: 0-65535, **default**= 0

See Also: ---

Details

The [on_sock_data_arrival](#)^[342] event is generated once the RX buffer is not empty, i.e. there is data to process. There may be only one on_ser_data_arrival event for each socket waiting to be processed in the event queue. Another on_sock_data_arrival event for the same socket may be generated only after the previous one is handled.

If, during the on_sock_data_arrival event handler execution, not all data is extracted from the RX buffer, another on_sock_data_arrival event is generated

immediately after the `on_sock_data_arrival` event handler is exited.

.Send Method

Function:	For the selected socket (selection is made through sock.num) commits (allows sending) the data that was previously saved into the TX buffer using the sock.setdata method.
Syntax:	<code>sock.rxbuffrq</code>
Returns:	---
See Also:	---

Details

You can monitor the sending progress by checking the [sock.txlen](#) property or using the [sock.notifysent](#) method and the `on_sock_data_sent` event.

.Setdata Method

Function:	For the selected socket (selection is made through sock.num) adds the data passed in the txdata argument to the contents of the TX buffer.
Syntax:	<code>ser.setdata(byref txdata as string)</code>
Returns:	---
See Also:	---

Part	Description
txdata	The data to send; this data will be added to the contents of the TX buffer.

Details

If the buffer doesn't have enough space to accommodate the data being added then this data will be truncated. Newly saved data is not sent out immediately. This only happens after the [sock.send](#) method is used to commit the data. This allows your application to prepare large amounts of data before sending it out.

Total amount of newly added (uncommitted) data in the buffer can be checked through the [sock.newtxlen](#) setting.

.Setsendinband Method

Function:	For the selected socket (selection is made through sock.num) puts the data into the RPL buffer. This method also commits the data.
Syntax:	sock.setsendinband(byref data as string)
Returns:	---
See Also:	---

Part	Description
data	The data to send (String).

Details

This method is different from the TX buffer for which two separate methods- [sock.setdata](#) and [sock.send](#) are used to store and commit the data. For the RPL buffer you store and commit the data with a single `sock.setsendinband` method.

It is the responsibility of your application to properly encapsulate outgoing messages with escape sequence ("EC OC", see the [sock.escchar](#) property) and the end character (see the [sock.endchar](#) property). When adding the data to the RPL buffer make sure you are adding entire inband message at once- you are not allowed to do this "in portions"!

Sinkdata Property

Function:	For the currently selected socket (selection is made through sock.num) specifies whether the incoming data should be discarded.
Type:	Enum (yes_no, byte)
Value Range:	0- NO (default): normal data processing. 1- YES: discard incoming data.
See Also:	Sinking Data

Details

Setting this property to 1- YES causes the socket to automatically discard all incoming data without passing it to your application. The [on_sock_data_arrival](#) event will not be generated, reading [sock.rrlen](#) will always return zero, and so on. No data will be reaching its destination even in case of [buffer redirection](#), [Inband commands](#), however, will still be extracted from the incoming data stream and processed. [Sock.connectiontout](#) and [sock.toutcounter](#) will work correctly as well.

.Splittcppackets Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) specifies whether the program will have additional control over the size of TCP packets being received and transmitted.
Type:	Enum (no_yes, byte)
Value Range:	0- NO (default): regular processing of TCP data in which your program does not care about the size of individual TCP data packets being transmitted and received. 1- YES: Additional features are enabled that allow your program to know the size of each incoming TCP packet and also control the size of each outgoing TCP packet.
See Also:	"Split Packet" Mode of TCP Data Processing ^[306]

Details

When this property is set to 1- YES your program gets an additional degree of control over TCP. For incoming TCP data, the program can know the size of individual incoming packets (this will be reported by the [on_sock_tcp_packet_arrival](#)^[341] event).

For outgoing TCP data, no packet will be sent out at all unless entire contents of the TX buffer can be sent. Therefore, by executing [sock.send](#)^[353] and waiting for [sock.txlen](#)^[364]=0 your program can make sure that the packet sent will have exactly the size you needed.

The property is only relevant when the [sock.inbandcommands](#)^[337]= 0- NO. With inband commands enabled, the socket will always behave as if the [sock.splittcpackets](#)= 0- NO. The program won't be able to change the value of this property when the socket is not idle ([sock.statesimple](#)^[358]<> 0- PL_SSTS_CLOSED).

Notice, that sending out TCP data and waiting for the [sock.txlen](#)^[364]=0 significantly diminishes your TX data throughput. This is because each send will be waiting for the other end to confirm the reception of data.

Also, with [sock.splittcpackets](#)= 1= YES make sure that you are not sending more data than the size of the RX buffer on the other end. If this happens, no data will ever get through because your side will be waiting for the chance to send out all TX data at once, and the other end won't be able to receive this much data in one piece.

Also, attempting to send the packet with size exceeding the "maximum segment size" (MSS) as specified by the other end will lead to data fragmentation! The socket will never send any TCP packet with the amount of data exceeding MSS.

.State R/O Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) returns "detailed" current socket state.
------------------	---

Type: Enum (pl_sock_state, byte)

Value Range:

0- PL_SST_CLOSED (**default**): Connection is closed (and haven't been opened yet, it is a post-powerup state). Applied both to UDP and TCP.

1- PL_SST_CL_PCLOSED: Connection is closed (it was a passive close). Applies only to TCP.

2- PL_SST_CL_ACLOSED: Connection is closed (it was an active close by the application). Applies only to TCP.

3- PL_SST_CL_PRESET_POPENING: Connection is closed (it was a passive reset during a passive open). Applies only to TCP.

4- PL_SST_CL_PRESET_AOPENING: Connection is closed (it was a passive reset during an active open). Applies only to TCP.

5- PL_SST_CL_PRESET_EST: Connection is closed (it was a passive reset while in "connection established" state). Applies only to TCP.

6- PL_SST_CL_PRESET_PCLOSING: Connection is closed (it was a passive reset while performing a passive close). Applies only to TCP.

7- PL_SST_CL_PRESET_ACLOSING: Connection is closed (it was a passive reset while performing an active close). Applies only to TCP.

8- PL_SST_CL_PRESET_STRANGE: Connection is closed (it was a passive reset, no further details available). Applies only to TCP.

9- PL_SST_CL_ARESET_CMD: Connection is closed (it was an active reset issued by the application). Applies only to TCP.

10- PL_SST_CL_ARESET_RE_PO: Connection is closed (it was an active reset issued because of excessive retransmission attempts during a passive open). Applies only to TCP.

11- PL_SST_CL_ARESET_RE_AO: Connection is closed (it was an active reset issued because of excessive retransmission attempts during an active open). Applies only to TCP.

12- PL_SST_CL_ARESET_RE_EST: Connection is closed (it was an active reset issued because of excessive retransmission attempts while in "connection established" state). Applies only to TCP.

13- PL_SST_CL_ARESET_RE_PCL: Connection is closed (it was an active reset issued because of excessive retransmission attempts during a passive close). Applies only to TCP.

14- PL_SST_CL_ARESET_RE_AC: Connection is closed (it was an active reset issued because of excessive retransmission attempts during a passive open). Applies only to TCP.

15- PL_SST_CL_ARESET_TOUT: Connection is closed (it was an active reset caused by connection timeout, i.e. no data was exchanged for `sock.connectiontout` number of seconds). Applies only to TCP.

16- PL_SST_CL_ARESET_DERR: Connection is closed (it was an active reset caused by a data exchange error).
Applies only to TCP.

17- PL_SST_CL_DISCARDED_CMD: Connection is closed (it was discarded by the application). Applies both to UDP and TCP.

See Also: ---

Details

This property tells the "detailed" current state of the socket, as opposed to the [sock.event](#)^[332] property that returns the "detailed" state at the moment of the [on_sock_event](#)^[343] event generation.

Another set of read-only properties- [sock.eventsimple](#)^[332] and [sock.statesimple](#)^[358]- return "simplified" socket states.

.Statesimple R/O Property

Function: For the currently selected socket (selection is made through [sock.num](#)^[340]) returns "simplified" current socket state.

Type: Enum (pl_sock_state_simple, byte)

Value Range: 0- PL_SSTS_CLOSED (**default**): Connection is closed. Applies both to UDP and TCP.

1- PL_SSTS_ARP: ARP resolution is in progress (it is an active open). Applies both to UDP and TCP.

2- PL_SSTS_PO: Connection is being established (it is a passive open). Applies only to TCP.

3- PL_SSTS_AO: Connection is being established (it is an active open). Applies only to TCP.

4- PL_SSTS_EST: Connection is established. Applies both to UDP and TCP.

5- PL_SSTS_PC: Connection is being closed (it is a passive close). Applies only to TCP.

6- PL_SSTS_AC: Connection is being closed (it is an active close). Applies only to TCP.

See Also: ---

Details

This property tells the current "simplified" state of the socket, as opposed to the [sock.eventsimple](#)^[332] property that returns the "simplified" state at the moment of the [on_sock_event](#)^[343] event generation.

Another set of read-only properties- [sock.event](#)^[332] and [sock.state](#)^[355]- return "detailed" socket states.

.Targetbcast Property

Function:	For the currently selected socket (selection is made through sock.num) specifies whether this port will be sending its outgoing UDP datagrams as link-level broadcasts.
Type:	Enum (no_yes, byte)
Value Range:	0- NO: UDP datagrams will be sent as "normal" packets 1- YES: UDP datagrams will be sent out as link-level broadcast packets
See Also:	---

Details

This property is only relevant for UDP communications ([sock.protocol](#) = PL_SOCKET_PROTOCOL_UDP). When this property is set to 1- YES the socket will be sending out all UDP datagrams as broadcasts and incoming datagrams won't cause port switchover, even if the latter is enabled through the [sock.reconmode](#) property.

.Targetinterface Property

Function:	For the currently selected socket (selection is made through sock.num) selects the network interface through which an outgoing network connection will be established.
Type:	Enum (pl_sock_interfaces, byte)
Value Range:	Platform-specific. See the list of pl_sock_interfaces constants in the platform specifications.
See Also:	Establishing Outgoing Connections

Details

This property is not available on the [EM202/200 \(-EV\), DS202](#) platform.

.Targetip Property

Function:	For active opens on the currently selected socket (selection is made through sock.num) specifies the target IP to which the socket will attempt to connect to. For passive opens specifies, in certain cases, the only IP address from which an incoming connection will be accepted.
Type:	String.

Value Range: Any valid IP address, i.e. "192.168.100.40". Default= "0.0.0.0"

See Also: [sock.targetport](#)^[360], [sock.remoteport](#)^[348]

Details

For active opens, this property is only relevant at the moment of connection establishment.

For incoming connections, whether this property will matter or not is defined by the [sock.inconmode](#)^[338] property. When the sock.inconmode= 1- PL_SOCK_INCONMODE_SPECIFIC_IPPORT or 2- PL_SOCK_INCONMODE_SPECIFIC_IP_ANY_PORT only the host with IP matching the one set in the sock.targetip property will be able to connect to the socket.

Current IP on the "other side" of the connection can always be checked through the [sock.remoteip](#)^[347] read-only property.

.Targetport Property

Function: For active opens on the currently selected socket (selection is made through [sock.num](#)^[340]) specifies the target port to which the socket will attempt to connect to. For passive opens specifies, in certain cases, the only port from which an incoming connection will be accepted.

Type: Word

Value Range: 0-65535, **default**= 0

See Also: [sock.targetip](#)^[359], [sock.remoteip](#)^[347]

Details

For active opens, this property is only relevant at the moment of connection establishment.

For incoming connections, whether this property will matter or not is defined by the [sock.inconmode](#)^[338] property. When the sock.inconmode= 1- PL_SOCK_INCONMODE_SPECIFIC_IPPORT an incoming connection will only be accepted from the port matching the one set in the sock.targetport property.

Current port on the "other side" of the connection can always be checked through the [sock.remoteport](#)^[348] read-only property.

.Toutcounter R/O property

Function: For the currently selected socket (selection is made through [sock.num](#)^[340]) returns the time, in 0.5 second intervals, elapsed since the data was last send or received on this socket.

Type: Word

Value Range: 0-65535, **default**= 0

See Also:[Closing Connections](#)^[290]**Details**

This property is reset to 0 each time there is some data exchanged across the socket connection. The property increments at 0.5 second intervals while no data is moving through this socket.

If the [sock.connectiontout](#)^[330] is not at 0, this property increments until it reaches the value of the [sock.connectiontout](#) and the connection is terminated. The [sock.toutcounter](#) then stays at the value of [sock.connectiontout](#).

If the [sock.connectiontout](#) is at 0, the maximum value that the [sock.toutcounter](#) can reach is 1. That is, the [sock.toutcounter](#) will be at 0 after the data exchange, and at 1 if at least 0.5 seconds have passed since the last data exchange.

.Tx2buffrq Method

Function:	For the selected socket (selection is made through sock.num ^[340]) pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the TX2 buffer of the socket.
Syntax:	sock.tx2buffrq(numpages as byte) as byte
Returns:	Actual number of pages that can be allocated (Byte).
See Also:	sock.txbuffrq ^[362]

Part	Description
numpages	Requested numbers of buffer pages to allocate.

Details

The TX2 buffer is required when inband commands are enabled ([sock.inbandcommands](#)^[337]= 1- YES), without it the socket won't be able to TX data. Returns actual number of pages that can be allocated. Actual allocation happens when the [sys.buffalloc](#)^[217] method is used. Unlike for TX or RX buffers there is no property to read out actual TX2 buffer capacity in bytes. This capacity can be calculated as num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the [sock.tx2buffrq](#). "-16" is because 16 bytes are needed for internal buffer variables.

Buffer allocation will not work if the socket port to which this buffer belongs is not idle ([sock.statesimple](#)^[358] is not at 0- PL_SSTS_CLOSED) at the time when [sys.buffalloc](#) executes. You can only change buffer sizes of sockets that are idle.

.Tx2len R/O Property

Function:	For the selected socket (selection is made through sock.num ^[340]) returns the amount of data waiting to be sent out in the TX2 buffer.
Type:	Word
Value Range:	0-65535, default = 0 (0 bytes)
See Also:	sock.txlen ^[364]

Details

The TX2 buffer is needed to transmit outgoing TCP data when inband commands (messages) are enabled ([sock.inbandcommands](#)^[337]= 1- YES). If your application needs to make sure that all data is actually sent out from the socket then it must verify that both TX and TX2 buffers are empty.

.Txbuffrq Method

Function:	For the selected socket (selection is made through sock.num ^[340]) pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the TX buffer of the socket.
Syntax:	sock.txbuffrq(numpages as byte) as byte
Returns:	Actual number of pages that can be allocated (Byte).
See Also:	sock.tx2buffrq ^[361]

Part	Description
numpages	Requested numbers of buffer pages to allocate.

Details

Actual allocation happens when the [sys.buffalloc](#)^[217] method is used. The socket is unable to TX data if its TX buffer has 0 capacity. Actual current buffer capacity can be checked through the [sock.txbuffsize](#)^[363] which returns buffer capacity in bytes. Relationship between the two is as follows: `sock.txbuffsize=num_pages*256-16` (or =0 when `num_pages=0`), where "num_pages" is the number of buffer pages that was GRANTED through the `sock.txbuffrq`. "-16" is because 16 bytes are needed for internal buffer variables.

Buffer allocation will not work if the socket port to which this buffer belongs is not idle ([sock.statesimple](#)^[358] is not at 0- PL_SSTS_CLOSED) at the time when `sys.buffalloc` executes. You can only change buffer sizes of sockets that are idle.

.Txbuffsize R/O Property

Function:	For the currently selected socket (selection is made through sock.num ^[340]) returns current TX buffer capacity in bytes.
Type:	Word
Value Range:	0-65535, default = 0 (0 bytes).
See Also:	---

Details

Buffer capacity can be changed through the [sock.txbuffrq](#)^[362] method followed by the [sys.buffalloc](#)^[217] method.

The `sock.txbuffrq` requests buffer size in 256-byte pages whereas this property returns buffer size in bytes. Relationship between the two is as follows: `sock.txbuffsize=num_pages*256-16` (or =0 when `num_pages=0`), where "num_pages" is the number of buffer pages that was GRANTED through the `sock.txbuffrq`. "-16" is because 16 bytes are needed for internal buffer variables. The socket cannot TX data when the TX buffer has zero capacity.

.Txclear Method

Function:	For the selected socket (selection is made through sock.num ^[340]) clears (deletes all data from) the TX buffer.
Syntax:	sock.rxclear
Returns:	---
See Also:	---

Details

Invoking this method will have no effect when the socket is not closed ([sock.statesimple](#)^[358]<> 0- PL_SSTS_CLOSED).

.Txfree R/O Property

Function:	For the selected socket (selection is made through sock.num ^[340]) returns the amount of free space in the TX buffer in bytes.
Type:	Word
Value Range:	0-65535, default = 0 (0 bytes).
See Also:	---

Details

Notice, that the amount of free space returned by this property does not take into account any uncommitted data that might reside in the buffer (this can be checked via [sock.newtxlen](#)^[339]). Therefore, actual free space in the buffer is `sock.txfree - sock.newtxlen`. Your application will not be able to store more data than this amount.

To achieve asynchronous data processing, use the [sock.notifysent](#)^[340] method to get [on_sock_data_sent](#)^[342] event once the TX buffer gains required amount of free space.

.Txlen R/O Property

Function:	For the selected socket (selection is made through sock.num ^[340]) returns total number of committed bytes currently found in the TX buffer.
Type:	Word
Value Range:	0-65535, default = 0 (0 bytes).
See Also:	---

Details

The data in the TX buffer does not become committed until you use the [sock.send](#)^[353] method.

Your application may use the [sock.notifysent](#)^[340] method to get [on_sock_data_sent](#)^[342] event once the total number of committed bytes in the TX buffer drops below the level defined by the `sock.notifysent` method.

.Urlsubstitutes

Function:	A comma-separated list of filenames whose extensions will be automatically substituted for ".html" by the internal webserver of your device.
Type:	String
Value Range:	0-40 characters, default = ""
See Also:	URL Substitution ^[322]

Details

The substitution will be used only if the resource file with the requested file name is not included in the project directly.

For example, setting this property to "pix1.bmp" will force the webserver to actually process "pix1.html", but only if the file "pix1.bmp" is not found. Data output by the webserver to the browser will still look like a ".bmp" file. For this to work, the "pix1.html" must exist in the project.

This property allows programmatic generation of non-HTML files. In the above example it is possible to generate the BMP file through a BASIC code. There is no other way to do this, since only HTML files are parsed for BASIC code inclusions.

.Varbuffrq Method

Function:	For the selected socket (selection is made through sock.num ^[340]) pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the VAR buffer of the socket.
Syntax:	sock.varbuffrq(numpages as byte) as byte
Returns:	Actual number of pages that can be allocated (Byte).
See Also:	---

Part	Description
numpages	Requested numbers of buffer pages to allocate.

Details

The VAR buffer is the buffer that stores the HTTP request string. The socket is unable to process HTTP requests if its VAR buffer has 0 capacity.

Actual allocation happens when the [sys.buffalloc](#)^[217] method is used. Unlike for TX or RX buffers there is no property to read out actual VAR buffer capacity in bytes. This capacity can be calculated as num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the [sock.varbuffrq](#). "-16" is because 16 bytes are needed for internal buffer variables.

Buffer allocation will not work if the socket port to which this buffer belongs is not idle ([sock.statesimple](#)^[358] is not at 0- PL_SSTS_CLOSED) at the time when sys.buffalloc executes. You can only change buffer sizes of sockets that are idle.

The VAR buffer is only required when you plan to use this socket in the HTTP mode- see [sock.httpmode](#)^[334] property, also [sock.httpportlist](#)^[335].

IO Object



The io object controls I/O lines, ports, and interrupt lines of your device.

Overview 3.6.1

The I/O object controls your device's individual I/O lines and ports. Each port groups eight I/O lines. You can control the state of each line or entire port.

The list of available I/O lines and ports is platform-specific, i.e. it depends on your

device. Two enum constant sets -- pl_io_num and pl_io_port_num -- define the set of available lines and ports. You can find the declaration of these enums in the "Platform-dependent Constants" section of your platform documentation.

There are two different methods for controlling I/O lines (ports):

- [Method with pre-selection](#)^[366];
- [Method without pre-selection](#)^[367].

I/O lines of your device can work as both inputs and outputs -- [Controlling Output Buffers](#)^[368] topic explains this.

Some I/O lines can work as interrupts -- see [Working With Interrupts](#)^[369] for explanation. The list of available interrupt lines is platform-specific. The pl_int_num enum constant set defines the set of related variables. You can find the declaration of this enum in the "Platform-dependent Constants" section of your platform documentation.

On many devices, a number of I/O lines may be shared with inputs/outputs of special function blocks (serial ports, etc.). When a special function block is enabled, I/O lines it uses cannot (should not) be manipulated through the io object.

Line/Port Manipulation With Pre-selection

I/O line manipulation with pre-selection works like this: you first select the line you want to work with using the [io.num](#)^[372] property. You can then read and set the state of this line using the [io.state](#)^[375] property. On certain platforms, you can also enable/disable the output buffer of the line with the [io.enabled](#)^[370] property -- more on this in the [Controlling Output Buffers](#)^[368] topic. Here is a code example:

```
io.num=PL_IO_NUM_5 'select line #5
io.enabled=YES 'enable this line
io.state=HIGH 'set this line to HIGH
io.state=LOW 'now set this line to LOW
io.enabled=NO 'configure the line as input now
x=io.state 'read the state of the line
```

Same can be done with ports -- use [io.portnum](#)^[374], [io.portstate](#)^[375], and [io.portenabled](#)^[373] properties to achieve this:

```

io.portnum=2 'select port #2
io.portenabled=&hFF 'this means that every port line's output buffer will
be enabled (&hFF=&b1111111)
'output &h55: port lines 0, 2, 4, and 6 will be at HIGH, 4 remaining lines
will be at LOW (&h55=&b01010101)
io.portstate=&h55
'output another value
op.portstate=0
'configure the port for input and read its state
io.portenabled=0
x=io.portstate

```

This way of controlling the lines/ports of your device is good when you are going to manipulate the same line/port repeatedly. Performance is improved because you select the line/port once and then address this line/port as many times as you need.

Note that I/O line and port names are platform-specific and are defined by `pl_io_num` and `pl_io_port_num` enums respectively. The declarations for these enums can be found in the "Platform-dependent Constants" section of your platform documentation.

On many devices, a number of I/O lines may be shared with inputs/outputs of special function blocks (serial ports, etc.). When a special function block is enabled, I/O lines it uses cannot (should not) be manipulated though the `io` object.

Line/Port Manipulation Without Pre-selection

I/O line manipulation without pre-selection is good when you need to deal with several I/O lines at once. For this, use `io.lineset`^[372], `io.lineget`^[371], and `io.invert`^[374] methods. These methods require the line number to be supplied directly, as a parameter. Therefore, pre-selection with the `io.num`^[372] property is not necessary. Moreover, executing these methods leaves the `io.num` value intact.

Here is an example of a serialized clock/data output. The clock line is `PL_IO_NUM_0` and the data line is `PL_IO_NUM_1`. Notice how this is implemented -- clock line is preselected once, then set LOW and HIGH using the `io.state`^[375] property. Meanwhile, the data line is updated using the `io.lineset` method. The variable `x` supposedly carries a bit of data to be output (where `x` gets is data is not shown).

```

Dim f As Byte
Dim x As low_high

io.num=PL_IO_NUM_0 'pre-select the clock line
For f=0 To 7
    ... 'obtain the value of the next bit, put it into x (not shown)
    io.state=LOW 'set the clock line LOW (the clock line has already been
    preselected)
    io.lineset(PL_IO_NUM_1,x) 'output the next data bit
    io.state=HIGH 'set the clock line HIGH (the clock line has already been
    preselected)
Next f

```

Direct port manipulation is achieved using `io.portset`^[374] and `io.portget`^[373] methods.

Note that I/O line and port names are platform-specific and are defined by `pl_io_num` and `pl_io_port_num` enums respectively. The declarations for these enums can be found in the "Platform-dependent Constants" section of your platform documentation.

On many devices, a number of I/O lines may be shared with inputs/outputs of special function blocks (serial ports, etc.). When a special function block is enabled, I/O lines it uses cannot (should not) be manipulated through the io object.

Controlling Output Buffers

As far as the I/O line/port control goes, there are two kinds of Tibbo devices and corresponding platforms -- those without output buffer control, and those with output buffer control. You can find this information in the "Platform-dependent Programming Information" of your platform documentation.

On devices without the output buffer control each I/O line's output driver is always enabled. If you want to use this line as an input, set its state to HIGH. After that, you can read this line's state. If the line is left unconnected or is not being pulled low externally you will read HIGH. If the line is being pulled low externally you will read LOW. Pulling the line LOW externally while this line's output driver is at HIGH will do no damage to the line.

Here is a code example in which we wait for the line #1 to become LOW:

```
io.num=PL_IO_NUM_1 'select the line
io.state=HIGH 'set it to HIGH so we can use it as an input
While io.state=HIGH 'wait until the line is pulled LOW externally
Wend
```

On devices with explicit output buffer control, you need to define whether the line is an output (set `io.enabled`= 1- YES) or input (set `io.enabled`= 0- NO). Trying to read the line while it is in the output mode will simply return the state of the line's own output driver. Forcing the line externally while it works as an output may cause a permanent damage to the device. For this kind of devices, the above code must be modified to look like this:

```
io.num=PL_IO_NUM_1 'select the line
io.enabled=NO 'disable the output driver (line will function as an input
now)
While io.state=HIGH 'wait until the line is pulled LOW externally
Wend
```

Since ports consist of individual lines, the same applies to ports as well. What needs to be understood is that each port line can be configured as input or output individually. Hence, a particular port doesn't have to be "all outputs" or "all inputs". Here is an example where the lower half of the port lines is configured for output, and the rest of the lines serve as inputs:

```
'This is an example for devices with explicit output buffer control
io.portnum=PL_IO_PORT_1
io.portenabled= &h0F 'configure lower 4 lines as outputs, the rest will be
used as inputs
```

Some I/O lines are shared with inputs/outputs of special function blocks (serial ports, etc.). When a special function block is enabled, certain (not all) I/O lines it

uses may be configured for input or output automatically. For such lines, when the corresponding special function block is disabled, the state of the output buffer is restored automatically to what it used to be prior to enabling this function block.

Note that I/O line and port names are platform-specific and are defined by `pl_io_num` and `pl_io_port_num` enums respectively. The declarations for these enums can be found in the "Platform-dependent Constants" section of your platform documentation.

Working With Interrupts

Some platforms have a number of I/O lines that can work as interrupt inputs. To enable a particular interrupt line, select it using the `io.intnum`^[374] property and enable the line with the `io.intenabled`^[370] property:

```
'enable interrupt line #0
io.intnum=PL_INT_NUM_0
io.intenabled=YES
```

Once the line has been enabled, the change in this line's state will generate an `on io int`^[373] event. The `linestate` argument of this event is bit-encoded: each bit of the value represents one interrupt line. For the `PL_INT_NUM_0`, the corresponding bit is bit 0, for `PL_INT_NUM_1` -- bit 1, and so on. A particular bit of the `linestate` argument is set when the state change (from LOW to HIGH, or from HIGH to LOW) has been detected on the related interrupt line. Event handler for the `on io int`^[373] event can then determine what triggered the interrupt:

```
Sub On_io_int(linestate As Byte)
    'check if it is the interrupt line #0 that has caused the interrupt
    If linestate And &h01 <>0 Then
        'yes, interrupt has been triggered by the interrupt line #0
        ...
    End If
End Sub
```

Please, note that the word "interrupt" is used here is a somewhat loose sense. On traditional microcontrollers, interrupt line status change causes a near-instantaneous pause to the execution of the main code and a jump to an "interrupt routine". Hence, the term "interrupt" -- the execution of the main code gets interrupted.

With the `io` object, the interrupt line state change does not disrupt the execution of the any event handler or reordering of pending events in the event `queue`^[74]. The `on io int`^[373] event is added to the end of the event queue and is not handled until all earlier events are processed. Therefore, nothing is actually interrupted. Note also that there is no guaranteed interrupt response speed here -- the time between line state change detection and the execution of the `on io int`^[373] event handler depends on the number of prior events waiting in the queue and, hence, cannot be pre-determined with any certainty.

Further, there may be only one `on io int` event waiting in the event queue.

Another such event will not be generated unless the previous one is processed (this prevents the event queue from getting overflowed). Therefore, some short-lived state changes may remain undetected.

Bottom line: the "interrupts" of the io object should be viewed as a more convenient alternative to programmatic polling of the I/O lines.

Note that interrupt line names, such as "PL_INT_NUM_0" are defined by the pl_int_num enum, which is platform-specific. The declaration of this enum can be found in the "Platform-dependent Constants" section of your platform documentation.

Properties, Events, Methods

This section provides an alphabetical list of all properties, methods, and events of the io object.

.Enabled Property (Selected Platforms Only)

Function:	Sets/returns the state of the output buffer for the currently selected I/O line (selection is made through the io.num ^[372] property).
Type:	Enum (no_yes, byte)
Value Range:	0- NO (default): disabled, I/O line works as an input 1- YES: enabled, I/O line works as an output
See Also:	Controlling Output Buffers ^[368] , io.state ^[375]

Details

Depending on the platform, explicit configuration of the output buffers may or may not be required. See "Platform-dependent Programming Information" of your platform documentation. The property is not available on platforms that do not require explicit output buffer configuration.

.Intenabled Property

Function:	Enabled/disables currently selected interrupt line (selection is made through the io.intnum ^[374] property).
Type:	Enum (no_yes, byte)
Value Range:	0- NO (default): the interrupt line is disabled 1- YES: the interrupt line is enabled
See Also:	Working With Interrupts ^[369]

Details

Change of state of any enabled interrupt line leads to the [on_int_ev](#)^[373] event generation. State change of disabled interrupt lines produces no effect.

.Intnum Property

Function:	Sets/returns the number of currently selected interrupt line.
Type:	Enum (pl_int_num, byte)
Value Range:	Platform-specific. See the list of pl_int_num constants in the platform specifications.
See Also:	Working With Interrupts <small>[369]</small>

Details

Selected interrupt line can be enabled or disabled using the [io.intenabled](#) [370] property. State change on enabled interrupt lines causes [on io int](#) [373] event generation.

In order to work correctly as an interrupt on certain platforms, the line may need to be configured as an input -- see [Controlling Output Buffers](#) [368] for details.

.Invert Method

Function:	Inverts the state of the I/O line specified by the num argument.
Syntax:	io.invert(num as pl_io_num)
Returns:	---
See Also:	Line/Port Manipulation Without Pre-selection <small>[367]</small> , io.lineget <small>[371]</small> , io.lineset <small>[372]</small>

Part	Description
num	Platform-specific. See the list of pl_io_num constants in the platform specifications.

Details

No line pre-selection with the [io.num](#) [372] property is required and the value of the io.num will not be changed.

.Lineget Method

Function:	Returns the state of the I/O line specified by the num argument.
Syntax:	io.lineget(num as pl_io_num) as low_high
Returns:	Current line state as LOW or HIGH (low_high enum values)
See Also:	Line/Port Manipulation Without Pre-selection <small>[367]</small> , io.lineset <small>[372]</small> , io.invert <small>[371]</small>

Part	Description
num	Platform-specific. See the list of pl_io_num constants in the platform specifications.

Details

No line pre-selection with the [io.num](#)^[372] property is required and the value of the io.num will not be changed.

.Lineset Method

Function:	Sets the I/O line specified by the num argument HIGH or LOW as specified by the state argument.
Syntax:	io.lineset(num as pl_io_num, state as low_high)
Returns:	---
See Also:	Line/Port Manipulation Without Pre-selection ^[367] , io.lineget ^[371] , io.invert ^[371]

Part	Description
num	Platform-specific. See the list of pl_io_num constants in the platform specifications.
state	LOW or HIGH (low_high enum values).

Details

No line pre-selection with the [io.num](#)^[372] property is required and the value of the io.num will not be changed.

.Num Property

Function:	Sets/returns the number of currently selected I/O line.
Type:	Enum (pl_io_num, byte)
Value Range:	Platform-specific. See the list of pl_io_num constants in the platform specifications.
See Also:	Line/Port Manipulation with Pre-selection ^[366]

Details

Selects a particular I/O line to be manipulated through the [io.state](#)^[373] and [io.enabled](#)^[370] properties (the latter may or not be available on your platform). The list of available I/O lines is defined by the pl_io_num constant.

On_io_int Event

Function: Generated when the change of state on one of the enabled interrupt lines is detected.

Declaration: **on_io_int(linestate as byte)**

See Also: [Working With Interrupts](#) [369]

Part	Description
linestate	0-255. Each bit of this value corresponds to one interrupt line in the order that these lines are declared in the pl_int_num enum.

Details

Interrupt lines are enabled/disabled through the [io.intenables](#) [370] property. Another on_io_int event is never generated until the previous one is processed.

.Portenabled Property (Selected Platforms Only)

Function: Sets/returns the state of the output buffers for the currently selected I/O port (selection is made through the [io.portnum](#) [374] property).

Type: Byte

Value Range: **Default**= 0 (all eight I/O lines of the port are configured as inputs)

See Also: [Controlling Output Buffers](#) [368], [io.portstate](#) [375]

Details

Each I/O port groups eight I/O lines. Each bit of this property's byte value corresponds to one "member" I/O line. Setting the bit to 0 keeps the output buffer turned off, while setting the bit to 1 enables the output buffer.

Depending on the platform, explicit configuration of the output buffers may or may not be required. See "Platform-dependent Programming Information" of your platform documentation. The property is not available on platforms that do not require explicit output buffer configuration.

.Portget Method

Function: Returns the state of the I/O port specified by the num argument.

Syntax: **io.portget(num as pl_io_port_num) as byte**

Returns: 0-255. Each bit of this value corresponds to one member I/O line of the 8-bit port.

See Also: [Line/Port Manipulation Without Pre-selection](#)^[367], [io.portset](#)^[374]

Part	Description
num	Platform-specific. See the list of pl_io_port_num constants in the platform specifications.

Details

No line pre-selection with the [io.portnum](#)^[374] property is required and the value of the io.portnum will not be changed.

.Portnum Property

Function:	Sets/returns the number of currently selected I/O port.
Type:	Enum (pl_io_port_num, byte)
Value Range:	Platform-specific. See the list of pl_io_port_num constants in the platform specifications.
See Also:	Line/Port Manipulation With Pre-selection ^[366]

Details

Selects a particular I/O port to be manipulated through the [io.portstate](#)^[375] and [io.portenabled](#)^[373] properties (the latter may or not be available on your platform). Each port groups eight I/O lines. The list of available I/O ports is defined by the pl_io_port_num constant.

.Portset Method

Function:	Sets the I/O port specified by the num argument to the state specified by the state argument.
Syntax:	io.portset(num as pl_io_port_num, state as byte)
Returns:	---
See Also:	Line/Port Manipulation Without Pre-selection ^[367] , io.portget ^[373]

Part	Description
num	Platform-specific. See the list of pl_io_port_num constants in the platform specifications.
state	0-255. Each bit of this value corresponds to one member I/O line of the 8-bit port.

Details

No line pre-selection with the [io.portnum](#)^[374] property is required and the value of the io.portnum will not be changed.

.Portstate property

Function:	Sets/returns the state of the currently selected I/O port (selection is made through the io.portnum ^[374] property).
Type:	Byte
Value Range:	0-255. Default value is hardware-dependent.
See Also:	Line/Port Manipulation With Pre-selection ^[366] , io.portenabled ^[373]

Details

Each I/O port groups eight I/O lines. Each bit of this property's byte value corresponds to one "member" I/O line.

.State Property

Function:	Sets/returns the state of the currently selected I/O line (selection is made through the io.num ^[372] property).
Type:	Enum (low_high, byte)
Value Range:	0- LOW 1- HIGH (Default value is hardware-dependent)
See Also:	Line/Port Manipulation with Pre-selection ^[366] , io.enabled ^[370]

Details

Romfile Object



The romfile object allows you to access resource files that you have added to your project. Resource files appear in the "Resource Files" folder of your project tree. These are files that are not processed by the compiler in any way, just added to the compiled binary file.

You can use resource files to store some permanent data that is too large to fit in a constant or when it is just more convenient for you to handle this data as a separate file.

The [romfile.open](#)^[375] method allows you to open the file you need. Only one file can

be opened at any given time and there is no need to close it. The size of the file you have opened can be checked through the [romfile.size](#)^[379] read-only property.



If you attempt to open a non-existent file its size will be returned as 0. This is how you know that the file does not exist!

There is a file pointer, accessible through the [romfile.pointer](#)^[379] property. When you (re)open the file the pointer is at file position 1 (first character in the file). You read the data from the file using the [romfile.getdata](#)^[378] method. As you read from the file, the pointer is moving- each time by the number of characters you've just read out. You can also move the pointer to the new position by writing desired value into the romfile.pointer property.



The pointer is a variable of word type, therefore, you can only access up to 65534 bytes in each file. You can add larger files to the project but only first 65534 bytes of each file will be accessible from your application.

The [romfile.find](#)^[377] method is used to find desired substring in the resource file. This allows you to organize quick search for the file portion you need, which is very useful if you keep some table data in the resource file (again, you can only search within first 65534 bytes of the file).

For example, supposing you have the following data in the file <settings.txt>:

```
IP,4,4,127.0.0.1
Port,0,65535,1001
Protocol,0,1,0
...
...
```

This sample data represents the list of settings (user-definable parameters) that your program keeps somewhere in the non-volatile memory (i.e. [stor](#)^[380] object). Each line of the file describes the setting name, minimum value (length), maximum value (length), and default value.

Now, supposing you need to extract the string describing the "Port" setting. Here is the code:

```

dim w as word
dim s as string

romfile.open("settings.txt")

if romfile.size=0 then
    'File does not exist! Do something about it!
end if

'look for the 'port' substring
w=romfile.find(1,"Port",1)
if w=0 then
    'Line not found- do something about it!
end if
romfile.pointer=w 'set the pointer

'now look for the end of this line
w=romfile.find(w,chr(13),1) 'we assume that the CR (ASCII=13) will
definitely be encountered

'OK, now read
s=romfile.getdata(w-romfile.pointer)

's now contains desired line from the file <settings.txt>
...
...

```

Notice, that with the romfile object characters are counted from 1, not 0.

Sometimes, you will need to open the file not for the purpose of accessing it, but for the purpose of passing a reference to this file to another object. Such reference "pointer" is provided by the [romfile.offset](#)^[378] read-only property.

.Find Method

Function:	Locates Nth instance of a substring within a currently opened resource (ROM) file.
Syntax:	romfile.find(frompos as word, byref substr as string, num as word) as word
Returns:	File position at which the substr resides or 0 if the desired substring occurrence is not found
See Also:	Romfile Object ^[375]

Part	Description
frompos	Position in the ROM file from which the search will be conducted (first character of the file is at position 1, if you set this parameter to 0 it will be interpreted as 1).
substr	Substring to search for.
num	Instance number (occurrence) of the substring we are looking for.

Details



The romfile object can only access first 65534 bytes of each file. This is because the romfile.pointer is a variable of word type. This method will also work only within this limit.

.Getdata Method

Function:	Reads data from a currently opened resource (ROM) file.
Syntax:	romfile.getdata(maxinplen as byte) as string
Returns:	String containing a part of the ROM file's data
See Also:	Romfile Object <small>[375]</small>

Part	Description
maxinplen	Maximum input length -- number of characters to read from the file.

Details

Readout size is limited by three factors (whichever is smaller): capacity of the receiving string, the amount of remaining data in the file (from current position--see the [romfile.pointer](#) [379]), and maxinplen parameter. Current position is moved forward by the actual number of bytes that was read out.



The romfile object can only access first 65534 bytes of each file. This is because the romfile.pointer is a variable of word type. This method will also work only within this limit.

.Offset R/O Property

Function:	Returns the offset of the currently opened file.
Type:	Dword
Value Range:	Only limited by the size of compiled binary of your project.
See Also:	romfile.open <small>[379]</small>

Details

This read-only property returns the address of the base of the currently opened file within the compiled binary of your project. The property is used to pass the data of the file to some other object that may need this data. This way, the "separation of labor" is maintained between the objects. The romfile object opens the file and passes the pointer to this file, in the form of the romfile.offset property value, to another object that requires this information. The property has no other uses.

This property is not available on the [EM200/200, DS202](#) [134] platform.

.Open Method

Function:	Opens or re-opens a resource (ROM) file.
Syntax:	romfile.open(byref filename as string)
Returns:	---
See Also:	Romfile Object , romfile.size

Part	Description
filename	Name of the ROM file to open.

Details

Only one ROM file can be opened at any given time. If the file exists and is not empty the [romfile.pointer](#) is set to 0 each time you use the `romfile.open` method. If the file does not exist or is empty the `romfile.pointer` is set to 0. There is no method (or need) to explicitly close the ROM file.

.Pointer Property

Function:	Sets/returns current (pointer) position in the resource (ROM) file.
Type:	Word
Value Range:	See explanation below
See Also:	Romfile Object

Details

When the file is (re)opened with the [romfile.open](#) method, the pointer is reset to the first character of the file (position 1), except when the file is not found or contains no data, in which case the pointer will permanently be equal to 0. Pointer position cannot exceed file size (see the [romfile.size](#) property). When you read from the file using the [romfile.getdata](#) method the pointer is automatically moved forward by the number of characters that have been read out.



The romfile object can only access first 65534 bytes of each file. This is because the `romfile.pointer` is a variable of word type.

.Size R/O Property

Function:	Returns the size of currently opened resource (ROM) file.
Type:	Word

Value Range: 0-65535

See Also: [romfile.open](#)^[379]

Details

0 size is returned when the file does not exist or the file is empty.



The size will be truncated to 65534 even if the actual file is larger. This is because the romfile.pointer is a variable of word type.

Stor Object



The stor object provides access to the non-volatile (EEPROM) memory in which your application can store data that must not be lost when the device is switched off.

The [stor.size](#)^[383] read-only property tells you the amount of EEPROM memory offered by the stor object. The [stor.set](#)^[382] is used to write the data to the EEPROM, while the [stor.get](#)^[381] method is used to read the data from the EEPROM.

Here is a simple example how to save the IP address of your device in the EEPROM:

```
dim s as string
dim x as byte

s=ddval(net.ip) 'this way it will take less space in the EEPROM (only 4
bytes needed)
x= stor.set(s,0) 'write EEPROM

'check result
if x<>len(s) then
    'EEPROM write failure- do something about this!
end if
```

And here is how you read this data back from the EEPROM:

```
net.ip=ddstr(stor.get(0,4))
```

Notice, that with the stor object addresses are counted from 1, not 0. That is, the first memory location has address 1.

Special configuration area

The EEPROM IC of the device is also used to store certain configuration information required by the device (for details see "Platform-dependent Programming Information" topic inside your platform specifications section). Memory available to

your application equals the capacity of the IC minus the size of the special configuration area.

By default, when you access the first byte of the EEPROM you are actually accessing the first memory location above the special configuration area. One property -- [stor.base](#)^[381] -- returns the size of this offset. On startup, stor.base is equal to the size of the special configuration area, so your program can only access the memory above this area.

You can change the stor.base and access configuration area when you need. For example, you can change the MAC address this way- next time the device boots up it will start using newly set address.

.Base Property

Function:	Sets/returns the base address of the EEPROM from which the area available to your application starts.
Type:	Word
Value Range:	1-<actual memory capacity>, default = <size of special configuration area>+1
See Also:	Stor Object ^[380] , stor.size ^[383]

Details

By default, the value returned by this property is the address of the first EEPROM location just above the special configuration area (for details see "Platform-dependent Programming Information" topic inside your platform specifications section). For example, if the size of the special configuration area on your platform is 8 bytes then stor.base will return 9 by default.

This Default value makes sure that your application won't overwrite MAC by mistake. When you are accessing EEPROM memory using [stor.set](#)^[382] or [stor.get](#)^[381] methods, you specify the start address. Actual physical address you access is start_address+stor.base.

If your application needs to change some parameters in the configuration area you can set the stor.base to 1- this way you will have access to the entire memory.

.Getdata Method (previously .Get)

Function:	Reads data from the EEPROM.
Syntax:	stor.getdata(startaddr as word, len as byte) as string
Returns:	String contains the data read out from the EEPROM.
See Also:	Stor Object ^[380] , stor.set ^[382]

Part	Description
------	-------------

startaddr	The starting address in the EEPROM memory (addresses are counted from 1, if you set this parameter to 0 it will be interpreted as 1).
len	Maximum number of bytes to read.

Details

The len parameter defines the *maximum* number of bytes to read from the EEPROM. Actual amount of extracted data is also limited by the capacity of the receiving variable and the starting address (in relation to the memory capacity of the EEPROM). Memory capacity can be checked through the [stor.size](#)^[383] read-only property. Notice that when the stor.getdata executes, an offset equal to the value of [stor.base](#)^[384] is added to the startaddr. For example, if the stor.base returns 9 and you do stor.getdata(1,3) then you will actually be reading the data starting from physical EEPROM location 9. If you set the stor.base to 1 you will be able to access the EEPROM right from the physical address 1.

By default, the stor.base is set in such a way as to allow access to the EEPROM starting from the address just above the special configuration area of your device (for details on what this area actually stores see "Platform-dependent Programming Information" topic inside your platform specifications section). By setting the stor.base to 1 you are allowing access to the special configuration area.



With Tibbo Basic release V2, this method had to be renamed from .get to .getdata. This is because the period (".") separating "stor" from "getdata" is now a "true" part of the language, i.e. it is recognized as a syntax unit, not just part of identifier. Hence, Tibbo Basic sees "stor" and "get" as separate entities and "get" is a reserved word that can't be used.

.Setdata Method (previously .Set)

Function:	Writes data into the EEPROM.
Syntax:	stor.setdata(byref datatoset as string, startaddr as word) as byte
Returns:	Actual number of bytes written into the EEPROM.
See Also:	Stor Object ^[380] , stor.get ^[384] , stor.size ^[383]

Part	Description
datatoset	Data to write into the EEPROM.
startaddr	Starting address in the EEPROM from which the data will be stored (addresses are counted from 1, if you set this parameter to 0 it will be interpreted as 1).

Details

The operation has completed successfully if the value returned by this method

equaled the length of the datatoset string. If this is not the case then the write has (partially) failed and there may be two reasons for this: physical EEPROM failure or invalid startaddr (too close to the end of memory to save the entire string).

Notice that when the stor.setdata executes, an offset equal to the value of [stor.base](#)^[38] is added to the startaddr. For example, if the stor.base returns 9 and you do stor.setdata("ABC",1) then you will actually be reading the data starting from physical EEPROM location 9. If you set the stor.base to 1 you will be able to access the EEPROM right from the physical address 1.

By default, the stor.base is set in such a way as to allow access to the EEPROM starting from the address just above the special configuration area of your device (for details on what this area actually stores see "Platform-dependent Programming Information" topic inside your platform specifications section). By setting the stor.base to 1 you are allowing access to the special configuration area.



With Tibbo Basic release **V2**, this method had to be renamed from .set to .setdata. This is because the period (".") separating "stor" from "setdata" is now a "true" part of the language, i.e. it is recognized as a syntax unit, not just part of identifier. Hence, Tibbo Basic sees "stor" and "set" as separate entities and "set" is a reserved word that can't be used.

.Size R/O Property

Function:	Returns total EEPROM memory capacity (in bytes) for the current device.
Type:	Word
Value Range:	Platform-dependent
See Also:	Stor Object ^[380]

Details

Certain amount of EEPROM memory is occupied by the special configuration section (for details see "Platform-dependent Programming Information" topic inside your platform specifications section).

By default, special configuration area is not accessible to the application and is excluded from memory capacity reported by the stor.size. For example, if the EEPROM IC used by this platform has 2048 bytes of data, and the size of the special configuration memory is 8 bytes, then the stor.size will return 2040 by default. At the same time, the default value of [stor.base](#)^[38] property will be 9, which means that the EEPROM locations 1-8 are occupied by the special configuration area.

If you set the stor.base to 1 (for instance, to edit the MAC address), the stor.size will show the capacity of 2048. In other words, the number this property returns is actual_EEPROM_capacity-stor.base+1.

Pat Object



The pat object allows you to "play" signal patterns on up to five LED pairs, each pair consisting of a green and red LED.

The channel to work with is selected through the [pat.channel](#)^[385] property. The first channel (channel 0) is the primary channel of your system. It utilizes green and red status LEDs that are present on all external devices, boards, and some modules offered by Tibbo. All modules have SG and SR I/O lines that are meant for controlling external status LEDs. Note that when the Tibbo BASIC application is not running, green and red status LEDs are used to display various [status information](#)^[184].

The remaining four channels (channel 1-4) are identical in function, but use regular I/O lines of Tibbo devices. Moreover, [pat.greenmap](#)^[385] and [pat.redmap](#)^[387] properties allow you to flexibly map the green and red LED control lines of each channel to any I/O lines of the device.

The pattern you play can be up to 16 steps long. Each "step" can be either "-" (both LEDs off), "R" (red LED on), "G" (green LED on), or "B" (both LEDs on). You can also define whether the pattern will only execute once or loop and play indefinitely. Additionally, you can make the pattern play at a normal, double, or quadruple speed.

You load the new pattern to play with the [pat.play](#)^[388] method. If the pattern is looped it will continue playing until you change it. If the pattern is not looped it will play once and then the [on_pat](#)^[388] event will be generated. When the event handler is entered, the pat.channel property will be automatically set to the channel number for which the event was generated.

LED patterns offer a convenient way to tell the user what your system is doing. You can devise different patterns for different states of your device.

Here is a simple example in which we keep the green LED on at all times, except when the button is pressed, after which the green LED is turned off and the red LED blinks three times *fast*. Additionally, both green and red LEDs blink 4 times on startup. In this example we work on channel 0:

```

Sub On_sys_init
    pat.channel=0 'not really necessary since 0 is the default value for
    this property
    pat.play("B-B-B-B-", PL_PAT_CANINT)
End Sub

Sub On_button_pressed
    pat.play("*R-R-R-", PL_PAT_CANINT)
End Sub

Sub On_pat
    If pat.channel=0 Then 'not really necessary since we are not using
    any other channels
        pat.play("~G", PL_PAT_CANINT)
    End If
End Sub

```

In the above example, the power-up pattern is loaded inside the [on_sys_init](#)^[220] event handler. This is not a looped pattern, so once it finishes playing the [on_pat](#)^[388] event is generated and the "permanent" pattern "green LED on" is loaded inside this event's handler. This new pattern is looped (notice "~"). When the button is pressed, a fast pattern (notice "*") is loaded. This one makes the red LED

blink three times. Again, this is not a looped pattern, so after it finishes playing the `on_pat` event is generated and the "permanent green" pattern is loaded again.

.Channel Property

Function:	Selects/returns the LED channel (status LED pair) to work with.
Type:	Byte
Value Range:	The value of this property won't exceed 4 (even if you attempt to set higher value). Default = 0 (channel 0 selected).
See Also:	---

Details

Channels are enumerated from 0. All other properties, methods, and an event of this object relate to the currently selected channel. Note that this property's value will be set automatically when the event handle for the `on_pat`^[386] event is entered.

.Greenmap Property

Function:	For the selected LED channel (selection is made through the <code>pat.channel</code> ^[385] property), sets/returns the number of the I/O line that will act as a green LED control line.
Type:	Enum (pl_int_num, byte)
Value Range:	Platform-specific, see the list of pl_int_num constants in the platform specifications.
Default values:	Channel 0: (-1) (no mapping, read-only): the green status LED (control line) of Tibbo device is always used by this channel; Channels 1-4: PL_IO_NULL (non-existent line).
See Also:	Pat.redmap ^[387]

Details

Channel 0 is special -- its LED control lines can't be remapped. This is because channel 0 uses standard green and red status LEDs (they are called SG and SR). For channel 0, reading the property always returns (-1), and writing has no effect.

All other channels use regular I/O lines of Tibbo devices. Any I/O line can be selected to be the green control line of the selected channel. By default, all control lines are mapped to the non-existent line PL_IO_NULL. Remap as needed and don't forget to configure the selected I/O line as an output -- this won't happen automatically.

On_pat Event

Function: Generated when an LED pattern finishes playing.

Declaration: **on_pat**

See Also: [Pat.play](#) [386]

Details

This can only happen for "non-looped" patterns. Multiple on_pat events may be waiting in the event queue. When the event handler for this event is entered the [pat.channel](#) [385] property is automatically set to the channel for which this event was generated.

.Play Method

Function: Loads a new LED pattern to play on the currently selected LED channel (selection is made through the [pat.channel](#) [385] property).

Syntax: **pat.play(byref pattern as string, patint as pl_pat_int)**

Returns: ---

See Also: ---

Part	Description
pattern	Pattern string, can include the following characters: '-': both LEDs off 'R' or 'r': red LED on 'G' or 'g': green LED on 'B' or 'b': both LEDs on '~': looped pattern (can reside anywhere in the pattern string) '*': double-speed pattern (can reside anywhere in the pattern string). You can use this symbol twice to x4 speed. Adding even more '*' will not have any effect.
patint	Defines whether the pat.play method is allowed to interrupt another pattern that is already playing: 0- PL_PAT_NOINT: cannot interrupt 1- PL_PAT_CANINT: can interrupt)

Details

Maximum pattern length is 16 "steps". The [on_pat](#)^[386] event is generated once the pattern finishes playing. Looped patterns never finish playing and thus the event is never generated for them.

Note that channels 1-4 require you to map LED control lines. See [pat.greenmap](#)^[385] and [pat.redmap](#)^[387] properties for details.

.Redmap Property

Function: For the selected LED channel (selection is made through the [pat.channel](#)^[385] property), sets/returns the number of the I/O line that will act as a red LED control line.

Type: Enum (pl_int_num, byte)

Value Range: Platform-specific, see the list of pl_int_num constants in the platform specifications.

Default values:

Channel 0: **(-1)** (no mapping, read-only): the green status LED (control line) of Tibbo device is always used by this channel;

Channels 1-4: **PL_IO_NULL** (non-existent line).

See Also: [Pat.greenmap](#)^[385]

Details

Channel 0 is special -- its LED control lines can't be remapped. This is because channel 0 uses standard green and red status LEDs (they are called SG and SR). For channel 0, reading the property always returns (-1), and writing has no effect.

All other channels use regular I/O lines of Tibbo devices. Any I/O line can be selected to be the red control line of the selected channel. By default, all control lines are mapped to the non-existent line PL_IO_NULL. Remap as needed and don't forget to configure the selected I/O line as an output -- this won't happen automatically.

Beep Object



The beep object allows you to generate "beep" patterns using the beeper (buzzer) attached to the CO pin of your device. "CO" stands for "clock output" and its output can be actually used for anything, not just controlling beeper. Frequency available on the CO pin is defined by the [beep.divider](#)^[388] property.

When the pattern starts playing, the CO line becomes an output automatically. Therefore, you do not need to use the [io.enabled](#)^[370] property to configure this line as output. When the pattern stops playing, the line will return to the input/output and HIGH/LOW state that it had before the pattern started playing.

The pattern you play can be up to 16 steps long. Each "step" can be either "-" (buzzer off), or "B" (buzzer on). You can also define whether the pattern will only execute one or loop and play indefinitely. Additionally, you can make the

pattern play at "normal" or double speed.

You load the new pattern to play with the [beep.play](#)^[389] method. If the pattern is looped it will continue playing until you changed it. If the pattern is not looped it will play once and then the [on_beep](#)^[388] event will be generated.

Buzzer patterns offer a convenient way to tell the user what your system is doing. You can devise different patterns for different states of your device. You can use these patterns together with LED patterns generated by the [pat](#)^[384] object.

Here is a simple example in which we generate three beeps on power up of the device.

```
sub on_sys_init
    beep.play("B-B-B", PL_PAT_CANINT)
end sub
```

To obtain a permanent, non-stop square wave on CO pin load the following pattern:

```
beep.play("B~", PL_PAT_CANINT)
```

.Divider Property

Function:	Sets the frequency of the square wave output on the CO line.
Type:	Word
Value Range:	0-65535, default = 1 (divide by 2)
See Also:	---

Details

Actual frequency can be calculated as `base_frq/(2*beep.divider)`. Setting this property to 0 is equivalent to 65536 (i.e. actual frequency will be `base_frq/131072`).

`Base_frq` depends on your platform -- you will find this information in the "Platform-dependent Programming Information" topic inside your platform specifications section.

Most low-cost buzzers have resonant frequency at which they emit the loudest sound. Check the specifications for your buzzer and set the divider accordingly.

On_beep Event

Function:	Generated when buzzer pattern finishes "playing".
Declaration:	<code>on_beep</code>
See Also:	Beep Object ^[387] , beep.play ^[389]

Details

This can only happen for "non-looped" patterns. Multiple on_beep events may be waiting in the event queue.

.Play Method)3

Function:	Loads new beeper pattern to play.
Syntax:	pat.play(byref pattern as string, patint as pl_beep_int)
Returns:	---
See Also:	Beep Object ^[387]

Part	Description
pattern	Pattern string, can include the following characters: '-': the buzzer is off 'B' or 'b': the buzzer is on '~': looped pattern (can reside anywhere in the pattern string) '*': double-speed pattern (can reside anywhere in the pattern string). New in V1.1 : You can use this symbol twice and, thus, obtain x4 speed if necessary!
patint	Defines whether the beep.play method is allowed to interrupt another pattern that is already playing: 0- PL_BEEP_NOINT: cannot interrupt 1- PL_BEEP_CANINT: can interrupt

Details

Maximum pattern length is 16 "steps". The [on_pat](#)^[386] event is generated once the pattern finishes playing (looped patterns never finish playing).

RTC Object



The RTC object facilitates access to the real-time counter (RTC) of the device. The RTC is an independent hardware counter that has its own power input. When the backup battery is installed, the RTC will continue running even when the rest of the device is powered off. The RTC keeps track of elapsed days, minutes, and seconds, not actual date and time. This is why it is called the "counter", not "clock". This platform includes a set of convenient date and time conversion syscalls that can be used to transform RTC data into current date/time and back (see [year](#)^[21], [month](#)^[204], [date](#)^[19], [weekday](#)^[21], [daycount](#)^[192], [hours](#)^[195], [minutes](#)^[203], and [mincount](#)^[203]).

Two methods of the RTC object- [rtc.getdata](#)^[390] and [rtc.setdata](#)^[391] are used to read and set the counter value.

Supposing that your project has curr_year, curr_month, curr_date, curr_hours, curr_minutes, and curr_seconds variables, here is how you set and get the time:

```
dim curr_year, curr_month, curr_date, curr_hours, curr_minutes,
curr_seconds as byte
dim x,y as word
...
...
'set RTC now (data is supposed to be prepared in curr_ variables)
rtc.set(daycount(curr_year,curr_month,curr_date),mincount(curr_hours,curr_
minutes),curr_seconds)
...
...
'get RTC now- first get the daycount and mincount into x and y
rtc.getdata(x,y,curr_seconds)
'now convert daycount and mincount into date and time
curr_year=year(x)
curr_month=month(x)
curr_date=date(x)
curr_hours=hours(y)
curr_minutes=minutes(y)
```

There is also [rtc.running](#)^[391] read-only property that tells you whether the RTC is working.



Notice, that after the device powers up and provided that the backup power was not present at all times the RTC may be in the undetermined state and not work properly until the rtc.set method is executed at least once. Incorrect behavior may include failure to increment or incrementing at an abnormal rate. Rtc.running cannot be used to reliably check RTC state in this situation.

It is not necessary to use rtc.set if the backup power was present at all times while the device was off.

.Getdata Method (Previously .Get)

Function:	Returns current RTC data as the number of elapsed days, minutes and seconds.
Syntax:	rtc.getdata(byref daycount as word, byref mincount as word, byref seconds as byte)
Returns:	Number of elapsed days, minutes, and seconds. Notice that this is done indirectly, through byref arguments.
See Also:	RTC Object ^[389]

Part	Description
daycount	Number of elapsed days.
mincount	Number of elapsed minutes.
seconds	Number of elapsed seconds.

Details

Some platforms include a set of convenient date and time conversion functions that can be used to transform "elapsed time" values into current weekday, date, and time (see [weekday](#)^[21], [year](#)^[21], [month](#)^[204], [date](#)^[19], [hours](#)^[195], [minutes](#)^[203]).



When the RTC is powered up after being off (that is, device had not power AND no backup power for a period of time), it may not work correctly until you set it using [rtc.set](#)^[39] method. Incorrect behavior may include failure to increment or incrementing at an abnormal rate. Rtc.running cannot be used to reliably check RTC state in this situation.

It is not necessary to use rtc.set if the backup power was present while the device was off.



With Tibbo Basic release **V2**, this method had to be renamed from .get to .getdata. This is because the period (".") separating "rtc" from "getdata" is now a "true" part of the language, i.e. it is recognized as a syntax unit, not just part of identifier. Hence, Tibbo Basic sees "rtc" and "get" as separate entities and "get" is a reserved word that can't be used.

.Running R/O Property

Function: Returns current RTC state.

Type: Enum (no_yes, byte)

Value Range: 0- NO: RTC is not running.

1- YES: RTC is running.

See Also: [RTC Object](#)^[389]

Details

When this property returns 0- NO this typically is the sign of a hardware malfunction (for instance, RTC crystal failure).



When the RTC is powered up after being off (that is, device had not power AND no backup power for a period of time), it may not work correctly until you set it using [rtc.set](#)^[39] method. Rtc.running cannot be used to reliably check RTC state in this situation.

.Setdata Method (Previously .Set)

Function: Presets the RTC with a number of elapsed days , minutes , and seconds.

Syntax: **rtc.setdata(daycount as word, mincount as word, seconds as byte)**

Returns: ---**See Also:** [RTC Object](#)^[389]

Part	Description
daycount	Number of elapsed days.
mincount	Number of elapsed minutes.
seconds	Number of elapsed seconds.

Details

Some platforms includes a set of convenient date and time conversion functions that can be used to transform actual date into time into "elapsed time" values (see [daycount](#)^[192] and [mincount](#)^[203]).



Notice, that after the device powers up and provided that the backup power was not present at all times the RTC may be in the undetermined state and not work properly until the `rtc.set` method is executed at least once. Incorrect behavior may include failure to increment or incrementing at an abnormal rate. `Rtc.running` cannot be used to reliably check RTC state in this situation.

It is not necessary to use `rtc.set` if the backup power was present at all times while the device was off.



With Tibbo Basic release **V2**, this method had to be renamed from `.get` to `.getdata`. This is because the period (".") separating "stor" from "setdata" is now a "true" part of the language, i.e. it is recognized as a syntax unit, not just part of identifier. Hence, Tibbo Basic sees "stor" and "set" as separate entities and "set" is a reserved word that can't be used.

LCD Object



This is the display (`lcd`) object, it allows you to operate a display panel. With the `lcd` object you can:

- Use any display from a growing list of [supported controllers/panels](#)^[408].
- Inquire about the [properties](#)^[393] of selected controller/panel.
- Draw [pixels](#)^[395], [lines](#)^[396], and [rectangles](#)^[396]; [fill areas](#)^[396].
- [Print text](#)^[397] -- non-aligned, aligned, rotated, etc.
- Display [BMP images](#)^[404].

Overview 3.12.1

In this section:

- [Understanding Controller Properties](#)^[393]
- [Preparing the Display for Operation](#)^[395]
- [Working With Pixels and Colors](#)^[395]
- [Lines, Rectangles, and Fills](#)^[396]
- [Working With Text](#)^[397]
- [Displaying Images](#)^[404]
- [Improving Graphical Performance](#)^[405]
- [Supported Controllers/Panels](#)^[408]

Understanding Controller Properties

Display panels come in all types, shapes, and sizes. Tibbo supports a limited but growing number of [controllers/panels](#)^[408]. Each supported controller requires its own "mini-driver". Your application cannot specify the desired controller. Instead, controller type is selected through the Customize Platform dialog, accessible through the Project Settings dialog. Once this is done, a proper "mini-driver" is added to the executable binary during compilation.

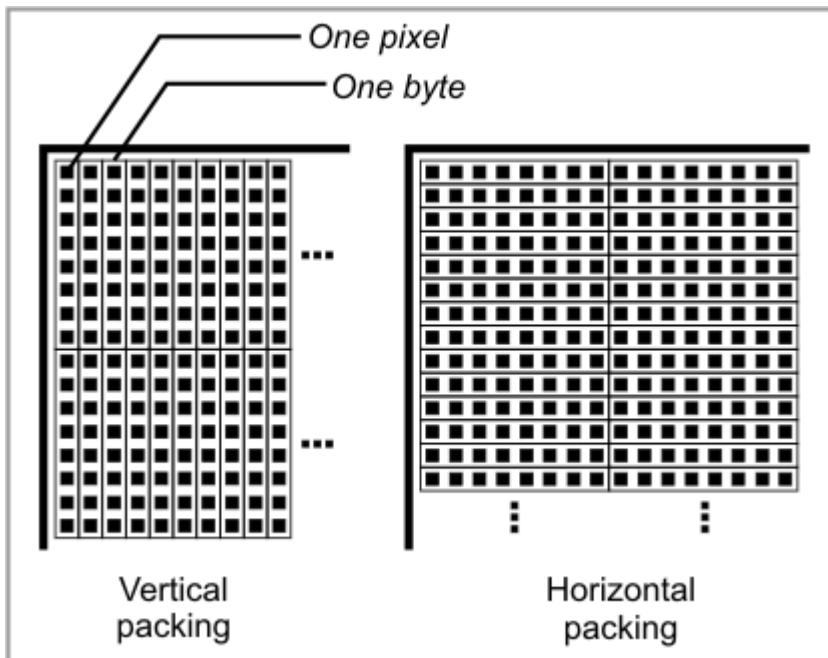
The `Lcd`. object is ready to work with a wide range of displays, some of which are not actually of LCD type. So, the proper name for this object should be "display object". Nevertheless, we have decided to keep this name -- typing "Lcd" is faster and the name has been in use for a while.

A number of R/O properties inform your application about the type and other vital parameters of the currently selected display:

[`Lcd.paneltype`](#)^[424] indicates whether this panel is of monochrome/grayscale or color type.

[`Lcd.bitsperpixel`](#)^[414] tells you how many bits in the display controller's memory are allocated for each display pixel. Of course, the more bits/pixel you get, the higher is your display quality.

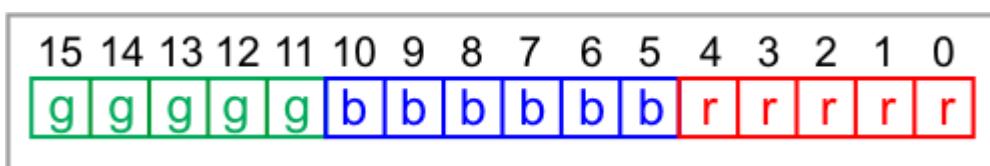
For displays with 1, 2, or 4 bits/pixel, a single byte of memory packs the data for 8, 4, or 2 pixels. [`Lcd.pixelpacking`](#)^[425] will inform you how pixels are packed into memory bytes: vertically, or horizontally (see the drawing below). The reason you may want to know this is to achieve faster [`text output`](#)^[397].



Color displays

For color displays (`lcd.paneltype= 1- PL_LCD_PANELTYPE_COLOR`), three additional R/O properties -- [lcd.redbits](#)^[427], [lcd.greenbits](#)^[420], and [lcd.bluebits](#)^[415] -- will indicate how many bits are available in each color channel, and also how three color fields are combined into a word describing the overall color of the pixel. You need to know this for setting pixels, as well as defining the foreground/background color used in drawing lines and rectangles, filling areas, and printing text (see [Working With Pixels and Colors](#)^[395]).

These three properties are of word type. Each 16-bit value packs two 8-bit parameters: number of bits per pixel for this color channel (high byte) and the bit position of the field in the color word (low byte). Supposing, `lcd.redbits=&h0500`, `lcd.bluebits=&h0605`, and `lcd.greenbits=&h050B`. You reconstruct the composition of the red, green, and blue bits in a word:



In this example, the red field is the first one on the right, followed by the blue field (this field starts from bit 5), then green field (starts from bit 11 or 7hB). You now also know that there are $5^2 = 32$ brightness levels for red and green, and 64 brightness levels for blue.

You can use this detailed information to select color values that will work correctly on all color displays, even those you haven't tested yet. Here is a useful example where you work out three constants -- `color_red`, `color_green`, and `color_blue` -- that will universally work for any color display.

```

Dim color_red, color_green, color_blue As word
...
lcd_red=val("&b"+strgen(lcd.redbits/&hFF,"1")+strgen(lcd.redbits And
&hFF,"0"))
lcd_green=val("&b"+strgen(lcd.greenbits/&hFF,"1")+strgen(lcd.greenbits And
&hFF,"0"))
lcd_blue=val("&b"+strgen(lcd.bluebits/&hFF,"1")+strgen(lcd.bluebits And
&hFF,"0"))

```

Now you can scientifically work out the constant for the white color:

```

...
lcd_white=lcd_red+lcd_green+lcd_blue

```



In reality, you don't have to bother calculating color_white like this. Just select the highest possible value (&hFFFF) and this will be your white.

Preparing the Display for Operation

Several steps need to be taken before the display will become operational. Some of these steps are display-specific. [Supported Controllers/Panels](#)⁴⁰⁸ section provides examples of startup code for each supported display.

Generally speaking, you need to take the following steps:

- Define which I/O lines and ports of your device control the display. This is done through the [Lcd.iomapping](#)⁴²² property.
- Configure some I/O lines/ports as outputs, as required for controlling your particular display. This is only necessary on platforms with explicit output buffer control. See "Platform-dependent Programming Information" of your platform documentation to determine if this step is required.
- Set the resolution of the display ([Lcd.width](#)⁴²³, [Lcd.height](#)⁴²⁴). These values depend on the panel, not the controller, so they cannot be detected automatically. Your application needs to set them "manually".
- Set [Lcd.rotated](#)⁴²⁸= 1- YES if you wish the display image to be rotated 180 degrees (that is, the display of your device is installed up side down).
- Set [Lcd.inverted](#)⁴²²= 1- YES if you need to invert the image on the display (may be required for certain panels).
- Enable the display by setting [Lcd.enabled](#)⁴¹⁶= 1- YES. This step will only work if your display is properly connected, correct display type is selected in your project, Lcd.iomapping is set property, and necessary I/O lines are configured as outputs. The [Lcd.error](#)⁴¹⁷ R/O property will indicate 1- YES if there was a problem enabling the display.
- Enable the backlight, if needed -- this is not related to the controller/panel itself, but is still a necessary step. Light it up, don't linger in the dark!

Working With Pixels and Colors

The [Lcd.setpixel](#)⁴²⁹ method allows you to set the color of a single pixel. The method accepts a 16-bit color word, and the interpretation of this word is controller/panel-specific (see [Understanding Controller Properties](#)³⁹³). In the following example, we determine the proper value for the green color and put a single green dot in the

middle of the display:

```
Dim color_green As word
...
lcd_green=val("&b"+strgen(lcd.greenbits/&hFF,"1")+
&hFF,"0"))
lcd.setpixel(lcd_green,lcd.width/2,lcd.height/2)
```

Lcd.setpixel is the only method that accepts the color directly. All other methods rely on two color properties: [lcd.forecolor](#)^[419] and [lcd.backcolor](#)^[414]. Each property is a 16-bit value, just like the one used by the lcd.setpixel. The forecolor is the color of the "drawing pen", and the backcolor is the color of the background. In the following example, we set the lcd.forecolor to the brightest color available, and the lcd.backcolor to the darkest color available:

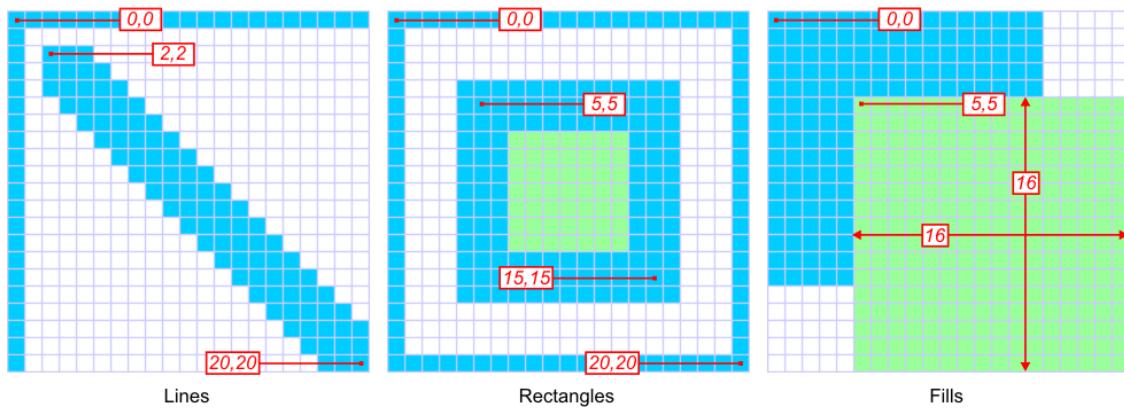
```
lcd.forecolor=&hFFFF
lcd.backcolor=0
```

The following methods are provided to output data onto the screen, and they all use the lcd.forecolor and, where necessary, the lcd.backcolor:

- [Lcd.line](#)^[422], [Lcd.verline](#)^[432], [Lcd.horline](#)^[421] -- the line is drawn using the forecolor.
- [Lcd.rectangle](#)^[427] -- the border is drawn using the forecolor, the internal area of the rectangle is not filled.
- [Lcd.filledrectangle](#)^[418] -- the border is drawn using the forecolor, the internal area is filled with the backcolor.
- [Lcd.fill](#)^[417] -- the entire area is filled with the forecolor.
- [Lcd.print](#)^[425] and [Lcd.printaligned](#)^[426] -- print text, where each "on" dot of each character is displayed in forecolor, while each "off" dot is displayed in backcolor.

The following two sections -- [Lines, Rectangles, and Fills](#)^[396], and [Working With Text](#)^[397] -- discuss the above methods in more details.

Lines, Rectangles, and Fills



Lines

The [lcd.line](#)^[422] method draws a line between any points. [lcd.verline](#)^[432] and [lcd.horline](#)^[423] draw vertical and horizontal lines correspondingly. Use the last two methods whenever possible because they work faster than generic lcd.line. The line is drawn using the color set in the [lcd.forecolor](#)^[419] property (see [Working With Pixels/Lines](#)^[395]), and the line width is defined by the [lcd.linewidth](#)^[423] property. In the following example, we draw a picture as shown above, on the left:

```
lcd.forecolor=color_blue 'we assume we have already set color_blue
lcd.verline(0,0,20) 'vertical line, width=1 (default)
lcd.horline(0,20,0) 'horizontal line, width=1 (default)
lcd.linewidth=3 'change the width
lcd.line(2,2,20,20) 'line at 45 degrees, width=3
```

Defining lcd.linewidth>1 (3 for one of the lines in the above example) creates "fatter" lines. Notice how two points of the line are drawn and where each specified coordinate actually is.

Rectangles

[Lcd.rectangle](#)^[427] draws an unfilled rectangle using lcd.forecolor as "pen" color, and pen width defined by the lcd.linewidth property. [Lcd.filledrectangle](#)^[418] will additionally paint the internal area using lcd.backcolor. Example and its result:

```
lcd.forecolor=color_blue 'we assume we have already set color_blue
lcd.backcolor=color_green 'we assume we have already set color_green
lcd.rectangle(0,0,20,20) 'width=1 (default)
lcd.linewidth=3
lcd.filledrectangle(5,5,15,15) 'width=3, filled with background color
```

Fills

[Lcd.fill](#)^[417] paints specified area with the lcd.forecolor:

```
lcd.forecolor=color_blue 'we assume we have already set color_blue
lcd.fill(0,0,16,16)
lcd.forecolor=color_green 'we assume we have already set color_green
lcd.fill(5,5,16,16)
```

Working With Text

[Lcd.print](#)^[425] and [Lcd.printaligned](#)^[426] display text. The text is printed using the selected font. This means you need to have at least one font file in your project (see how to [add](#)^[128] a file), and have this font selected before you can print anything.

[Lcd.setfont](#)^[428] is used to select the font:

```
romfile.open("Tibbo-5x7(VP).bin")
lcd.setfont(romfile.offset)
```

Note that lcd.setfont will return 1- NG if you try to feed it a wrong file!

Once the font has been successfully selected, the [lcd.fontheight](#)^[418] and [lcd.fontpixelpacking](#)^[419] R/O properties will tell you the maximum height of characters in this font and how pixel data is packed within the font file. The meaning of the first one is obvious. The meaning of the second one will become apparent after (and if) you read [Raster Font File \(TRF\) Format](#)^[400]. If you don't want to read this, it's OK too, we can just go straight to the summary:

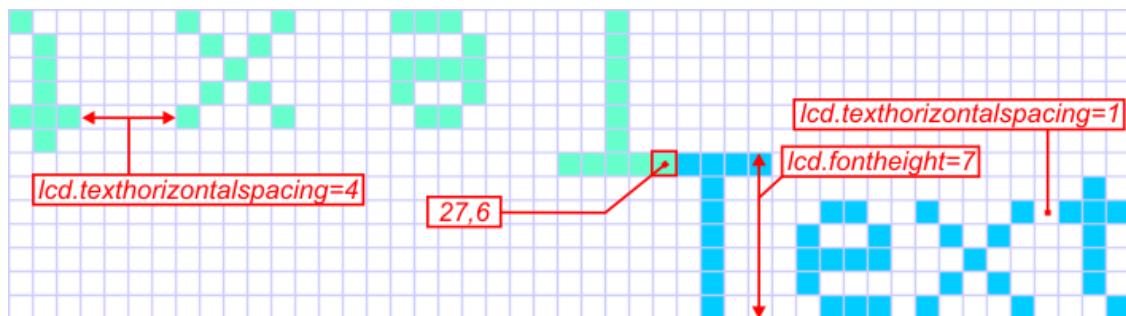
Fonts can be encoded horizontally or vertically, and the [lcd.fontpixelpacking](#) will tell you what type of font you have selected. You should only care about this for displays with [lcd.bitsperpixel](#)^[414]<8. if your display has [lcd.bitsperpixel](#)>8 then it doesn't matter what kind of font you use. If it is <8, then you are better off selecting a font for which [lcd.fontpixelpacking](#) is equal to [lcd.pixelpacking](#)^[425] (depends on selected controller/panel). You achieve better performance when these two properties are "aligned". It will also work if you select a font which is "perpendicular" to your display, but text printing might slow down considerably.

We typically offer two versions for each font, for example, "Tibbo-5x7(V).bin" and "Tibbo-5x7(H)". V stands for "vertical" and "H" stands "horizontal".

Non-aligned text

Once the font has been selected, you can start printing. You do this with the [lcd.print](#)^[425] method. This method always produces a single-line output. Two properties -- [lcd.textorientation](#)^[430] and [lcd.texthorizontalspacing](#)^[430] -- affect how your text is printed. The reference point of your text is at the top-left pixel of the output. X and Y arguments of [lcd.print](#) specify this corner, and the text is rotated "around" this pixel as well. Example:

```
lcd.print("Text",30,10)
lcd.textorientation=PL_LCD_TEXT_ORIENTATION_180
lcd.texthorizontalspacing=4
lcd.print("Text",30,10)
```



[Lcd.print](#) returns the total width of produced textual output in pixels. This can be very useful. Here is an example where you draw a frame around the text, and you want the frame size to be "just right":

```
x=lcd.print("Text",30,10)
lcd.rectangle(28,8,30+x+1,10+lcd.fontheight+1)
```

Sometimes, it is desirable to know the width of the text output before actual printing. [Lcd.getprintwidth](#)^[420] will tell you how many pixels will be taken by your

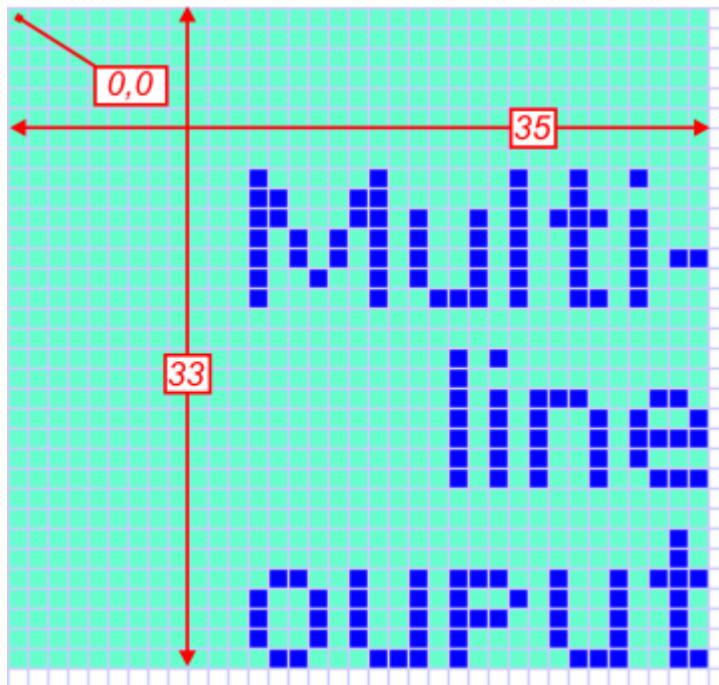
text horizontally (remember, text height is equal to [lcd.fontheight](#)^[418]). In the following example, we align the text output along the right side of the screen. This requires us to know how wide this output will be:

```
x=lcd.getprintwidth("Text")
x=lcd.print("Text",lcd.width-x,10)
```

Aligned text

The [lcd.printaligned](#)^[426] method outputs your text within a specified rectangular area. Four properties -- [lcd.textorientation](#)^[430], [lcd.textAlignment](#)^[429], [lcd.texthorizontalspacing](#)^[430], and [lcd.textverticalspacing](#)^[431] -- define how your text will be printed. To fit within the target area, the lcd. object will split the text into several lines as necessary. Only the text that can fit within the area will be displayed. You can add your own line breaks by using the ` character (ASCII code 96). Example:

```
lcd.textverticalspacing=2
lcd.textAlignment=PL_LCD_TEXT_ALIGNMENT_BOTTOM_RIGHT
lcd.printaligned("Multi-line text",0,0,35,33)
```



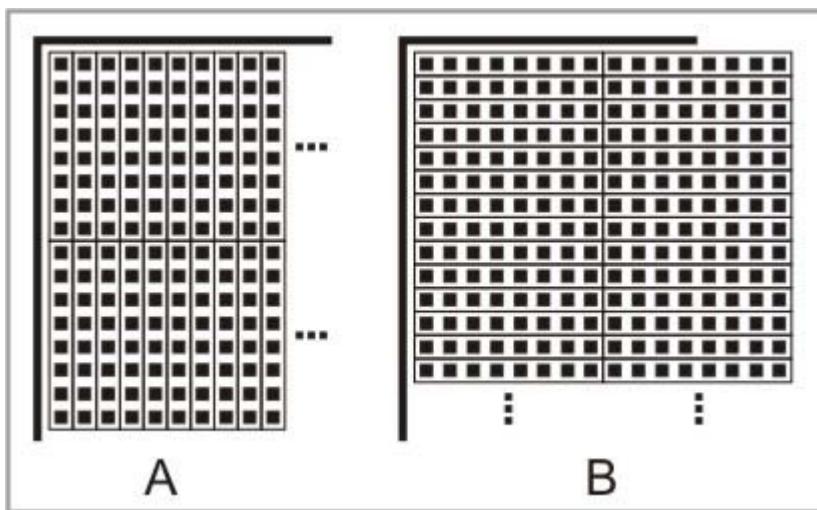
Note that lcd.printaligned returns the number of text lines that were produced.

Raster Font File Format

This topic details font file format used by the LCD object. The font file is a resource file that is added to your Tibbo Basic project. Like all other resource files "attached" to your project, font files are accessible through the [romfile](#)^[375] object. Use and interpretation of font file data, however, is the responsibility of the `Lcd` object. The Romfile object merely stores these files.

Tibbo font files have "TRF" (Tibbo Raster Font) extension. TRF file format was designed with the following considerations in mind:

- Ability to handle large character sets (such as those required for Chinese language support). Hence, the use of 16-bit character codes.
- 16-bit character sets usually have large "gaps" (i.e. areas of unused codes). The TRF format offers an efficient way to define which characters are included into the font file and allows to conduct very efficient character search within the file.
- Support for proportional fonts. Hence, each character's width is individually defined.
- Support for fonts with anti-aliasing. Anti-aliasing is achieved by adjusting the "intensity" (brightness) of individual pixels. In an anti-aliased font, each pixel of character bitmap is represented by 2 or more bits of data. Fonts without anti-aliasing just need 1 data bit/pixel because each pixel can only be ON or OFF. At the time of writing, TiOS supported only fonts with 1 bit/pixel.
- Support for vertical and horizontal character bitmap encoding. Displays with `Lcd.bitsperpixel`^[414]= 1, 2, or 4 pack 8, 4, or 2 pixels into a single byte of display memory. Problem is, some displays combine the pixels vertically (see drawing A below), and some -- horizontally (drawing B). Text output on such displays is more efficient if character bitmaps of the TRF file use the same direction of packing.



TRF file format

The TRF file consists of four data areas:

Data area	Description
-----------	-------------

Header	Contains various information such as the total number of characters in this font, character height, etc. Also contains the number of code groups in the code groups table (see below).
Code groups table	Contains descriptors of "code groups". Each code group contains information about a range of codes that has no "gaps" (i.e. unused codes in the middle). The font file can have as many code groups as necessary.
Bitmap offset table	Contains addresses (offsets) of all character bitmaps in the TRF file. In combination with the code groups table provides a way to find the bitmap of any specific character.
Bitmaps	Contains all bitmaps of each character included into the font file. The width of each bitmap is defined individually and is stored together with the bitmap.

Header format

The header has a fixed length of 16 bytes and stores the following information:

Offset *	Bytes	Data	Comment
+0	2	Num_o_f_chars	Total number of characters in this font file
+2	1	Pixels_per_byt e	0- eight pixels/byte (1 bit/pixel, no anti-aliasing); 1- four pixels/byte, 2- two pixels/byte, 3- one pixel/byte. Modes 1, 2, and 3 are currently not supported; these modes are for anti-aliasing and will be supported in the future.
+3	1	Orientation	0- pixels are grouped vertically; 1- pixels are grouped horizontally (irrelevant when pixels_per_byte =3)
+4	1	Height	Maximum character height in this font, in pixels.
+5	9	<Reserved>	Reserved for future use
+14	2	Num_o_f_grou ps	Number of entries in the code groups table

*With respect to the beginning of the file

Code groups table

This table has variable number of entries. This number is stored in the **num_of_groups** field of the header. Each code group represents a range of codes that contains no gaps (no unused character codes in the middle). For example, supposing that we have a font that only contains characters '0'-'9' and 'A'-'Z'. This means that this font file will contain two groups of codes: 0030H through 0039H ('0'-'9') and 0041H through 005AH ('A'-'Z').

Each entry in the code groups table is 8 bytes long and has the following format:

Off set *	By tes	Data	Comment
+0	2	Start_code	The first code in the group
+2	2	Num_codes	Number of individual character codes in this group
+4	4	Bitmap_addr_offset	Address (that falls within the bitmap offset table and is given with respect to the beginning of the file) at which the address of the bitmap of the first character in the code group is stored.

* With respect to the beginning of a particular table entry

For the above example the code groups table will have two entries:

Start_code	Num_codes	Bitmap_offset
0030H	000AH	00000020H
0041H	001AH	00000048H

Here is how the above data was calculated. Start codes are obvious. Group one starts with code 0030H because this is the character code of '0'. The second group starts with the character code of 'A'. It is also easy to fill out the number of codes in each group: 10 (000AH) for '0'-'9' and 26 (001AH) for 'A'-'Z'.

Bitmap_addr_offset calculation is explained in the next section.

Bitmap offset table

This table has the same number of entries as the total number of characters included in the font file. Each entry consists of one field -- a 32-bit offset of a particular bitmap with respect to the beginning of the font file. Now you can see how we were able to calculate the data for the **bitmap_addr_offset** field of the code groups table. The header of the font file has a fixed length of 16 bytes. There are two code groups in our example, so the code groups table occupies $8 \times 2 = 16$ bytes. This means that the bitmap offset table starts from address $16 + 16 = 32$ (0020H). Hence, the first entry in the code groups table points at address 0020H. The first code group contains 10 characters ('0'-'9'). These will "occupy" 10 entries in the bitmap offset table, which results in $10 \times 4 = 40$ bytes. Hence, the **bitmap_addr_offset** field for the second code group is set to $32 + 40 = 72$ (0048H).

Bitmaps

Each bitmap starts with a single byte that encodes the width of the bitmap in pixels, followed by the necessary number of bytes representing this bitmap. Depending on the **pixels_per_byte** field of the header, each byte of data may encode just one or several pixels. Additionally, when using more than 1 pixel-per-

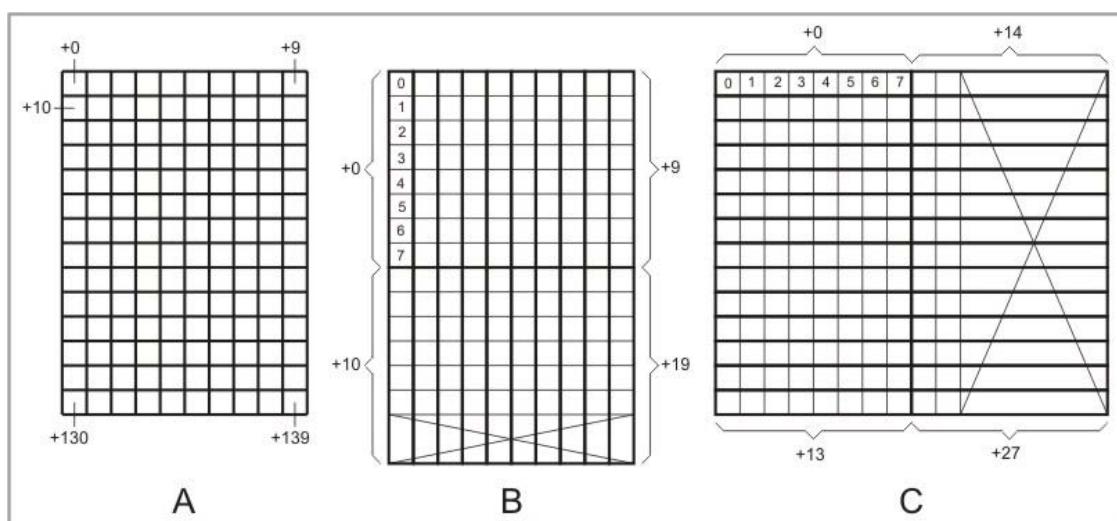
byte encoding, the **orientation** field of the header defines whether pixels are combined horizontally or vertically.

The drawings below illustrate how character bitmaps are stored in the font file. As an example, characters of 10x14 size (in pixels) are used. Drawing A is for one pixel/byte encoding, drawing B -- for 8 pixels/byte with vertical orientation, C -- for 8 pixels/byte with horizontal orientation. Notice that for cases B and C a portion of some bytes used to store the bitmaps is unused. Offsets of bytes relative to the beginning of the bitmap data are shown with a '+' sign.

Bitmap A takes 140 bytes. The first byte (+0) represents the pixel at the top left corner of the bitmap. Subsequent bytes represent all other pixels and the order is "left-to-right, top-to-bottom".

Bitmap B takes 20 bytes. The first byte encodes 8 vertically arranged pixels at the top left corner of the bitmap. Subsequent bytes represent all other pixel groups and the order is "left-to-right, top-to-bottom". There are 2 rows of bytes, and bits 6 and 7 of each byte in the second row are unused.

Bitmap C takes 28 bytes. The first byte encodes 8 horizontally arranged pixels at the top left corner of the bitmap. Subsequent bytes represent all other pixel groups and the order is "top-to-bottom, left-to-right". There are 2 columns of bytes, and bits 2-7 of each byte in the second column are unused.



Searching for a character bitmap

Here is how a target character bitmap is found within the font file. Again, we are using the example of the font file that contains characters '0'-'9' and 'A'-'Z'.

Supposing, we need to find the bitmap of character 'C' (code 0043H).

First, we need to see which code group code 004AH belongs to. We read the **num_of_groups** field of the header to find out how many code groups are contained in the font file. The field tells us that there are two groups.

Next, we start reading the code groups table (located at file offset +00000010H), entry by entry, in order to determine which code group the target character belongs to. The first group starts from code 0030H and contains 10 characters. Therefore, target character doesn't belong to it. The second group starts from code 0041H and contains 26 characters. The target code is 0043H. Therefore, target character belongs to this second group.

Next, we find the corresponding entry in the bitmap offset table. For this, we do a

simple calculation:

bitmap_offset + (desired_code - **start_code**) * 4: 00000048H + (0043H-0041H)
*4= 00000050H.

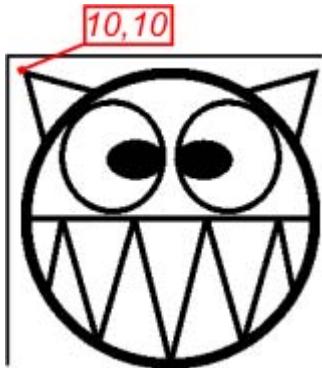
Next, we read a 32-bit value at file offset 00000050H. This will tell us the file offset at which the target bitmap is stored.

At this file offset, the first byte will be the width of the bitmap in pixels. Based on this width and also **height**, **pixels_per_byte** and **orientation** fields of the header we can calculate the number of bytes in the bitmap. For example, supposing that **height** = 14, **pixels_per_byte** = 0 (8 pixels/byte), and **orientation** = 0 (pixels are grouped vertically). Also, let's suppose that the width of the target character is 10 pixels. In this case the bitmap will occupy 20 bytes, as shown on the drawing B above. Two bits of each byte in the second byte row will be unused.

Displaying Images

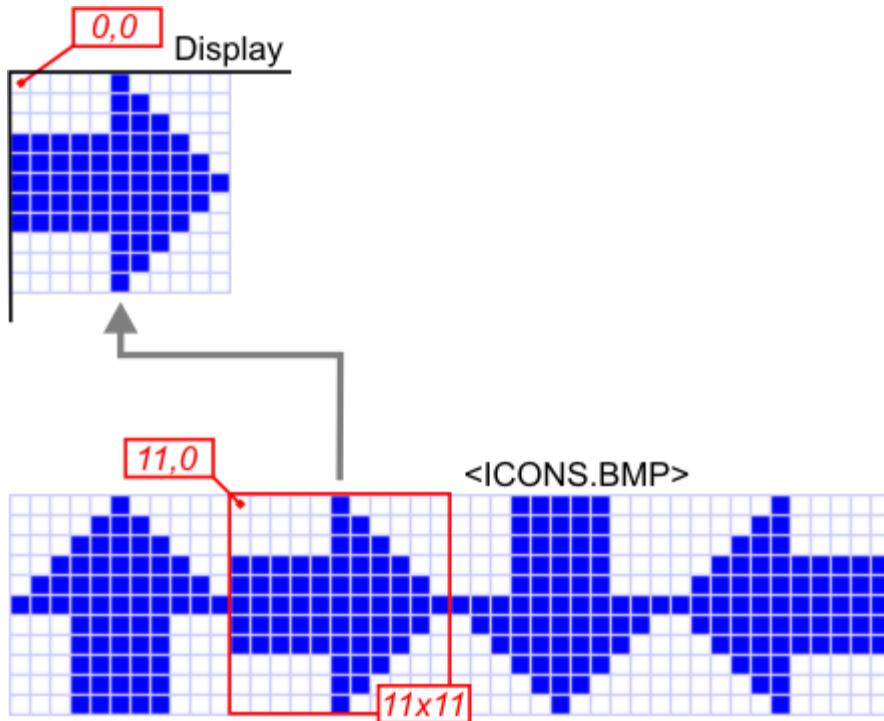
[Lcd.bmp](#)^[415] outputs a full image, or a portion of the image from a BMP file. Naturally, this file must be present in your project (see how to [add](#)^[128] a file). Here is a simple example of how to display an image:

```
romfile.open("mad_happiness.bmp")
lcd.bmp(romfile.offset,0,0,0,0,65535,65535) 'see how we set maxwidth and
maxheight to 65535 (max value). This way we specify that the entire image
should be displayed (if only it can fit on the screen)
```



One powerful feature of the `Lcd.bmp` method is that you can selectively display a portion of the image stored in a BMP file. This opens up two interesting possibilities. First, you can scroll around a large file image by specifying which part you want to see. Second, you can combine several images into one large file and display required portions when needed. For example, if your project requires several icons you can put them all into a single "strip". This improves performance because you don't have to open a separate file to display each icon:

```
romfile.open("icon_strip.bmp")
lcd.bmp(romfile.offset,0,0,11,0,11,11) 'display the second icon (right arrow)
```



Note that only 2-, 16-, and 256-color modes are currently supported and the lcd.bmp will return 1- NG if you try to display any other type of BMP file. Compressed BMP files will be rejected too.

The method takes into account the type of the currently selected controller/panel. The method will check the values of [lcd.paneltypes](#)^[424], [lcd.bitsperpixel](#)^[414], [lcd.redbits](#)^[427], [lcd.greenbits](#)^[428], and [lcd.bluebits](#)^[415] (explained in [Understanding Controller Properties](#)^[393]) and produce the best output possible for the selected display.

Improving Graphical Performance

Nobody likes sluggish products, and the way you work with your display can greatly influence *perceived* performance of your product. We say "perceived" because very often this has nothing to do with the actual speed at which your device completes required operations.

Group display-related operations together

The most important aspect of your application's performance (related to the display) is how fast the data on the display appears to have changed. Interestingly, what matters most is not the total time it takes to prepare and show the new data, but the "togetherness" at which all new displayed items pop up on the screen.

Let's illustrate this point. Here are two code examples that do the same and take roughly the same time to execute. We calculate and print two values. In the first example, we calculate and print value 1, then calculate and print value 2:

```

Dim dw1,dw2 As dword
...
'Note so good! The user will see a noticeable delay between the first and
the second print.
lcd.print(str(dw1*100/200),0,0)
lcd.print(str(dw2*100/200),0,10)

```

In the second example we calculate values 1 and 2 first, then print them together:

```

Dim dw1,dw2 As dword
Dim s1,s2 As String
...
'Much better. The user will have a feeling that both values were
calculated and printed instantly!
s1=str(dw1*100/200)
s2=str(dw2*100/200)
lcd.print(s1,0,0)
lcd.print(s2,0,10)

```

Testing both examples shows that the perceived performance of the second code snippet is much better, while, in fact, the total working time of the processor was roughly the same. Why is there a difference? Because the output of the two values in the second example was spaced closer!

Bottom line: keep all display output as close together as possible. Pre-calculate everything first, then display all your items "at once".

Use display locking

No matter how hard you try to group all display-related output together, executing `Lcd`. object's methods one after another may still take considerable time. Perceived performance can be improved on displays that allow you to "lock" display picture, change display data, then unlock the display again. With this approach, the user will see all changes appear instantly! Not all displays are suitable for this.

Typically, this works well for TFT panels which continue to display the image for several seconds after the "refresh" was disabled. Other display types are not suitable for locking. We have provided locking-related info for each [supported controller/panel](#)^[408].

The display is locked/unlocked using the [`Lcd.lock`](#)^[423] and [`Lcd.unlock`](#)^[431] methods. You can place `Lcd.lock` before the block of code that changes display data, and put `Lcd.unlock` at the end of this code block:

```

...
s1=str(dw1*100/200)
s2=str(dw2*100/200)
lcd.lock 'lock the display
lcd.print(s1,0,0)
lcd.print(s2,0,10)
lcd.unlock 'unlock the display
...

```

Display-related code is often nested, with one procedure calling another, and so

on. If you are using display locking, you should ideally place locks/unlocks in each related routine. A complication arises with regards to when to unlock the display. For example, supposing you have two subs: lcd_sub1 and lcd_sub2:

```
'some main routine that invokes lcd_sub1 and lcd_sub2
...
...
lcd_sub1
...
...
lcd_sub2
...
End Sub

'-----
Sub lcd_sub1 'calls sub2 too
lcd.lock
...
lcd_sub2
...
lcd.unlock
End Sub

'-----
Sub lcd_sub2 'called by sub1
lcd.lock
...
lcd.unlock
End Sub
```

Lcd_sub2 gets executed when invoked directly by the main code, and also when the lcd_sub1 is called. In the second case, the display should not be unlocked at the end of lcd_sub2 because the output is to be continued in lcd_sub1! And you know what? The display won't be unlocked because with the lcd object it is possible to nest locks/unlocks! In the following example, we do three consecutive locks and the display is locked right on the first one. We then do three consecutive unlocks, and the display is not unlocked until after the third one is executed:

```
...
lcd.lock 'lcd.lockcount=1, display is now locked
lcd.lock 'lcd.lockcount=2
lcd.lock 'lcd.lockcount=3
lcd.unlock 'lcd.lockcount=2
lcd.unlock 'lcd.lockcount=1
lcd.unlock 'lcd.lockcount=0, display is now unlocked
...
```

The lock/unlock mechanism maintains a counter, which can actually be read through the [lcd.lockcount](#)^[424] R/O property. Each invocation of lcd.lock increments the counter by 1, each lcd.unlock decrements it by 1. The display is only unlocked when the counter is at 0, and locked when the counter is >0. This allows you to nest display-related procedures and safely have lock/unlock in each one of them!

Supported Controllers/Panels

The following controllers/panels are currently supported:

- [Samsung S6B0108 \(Winstar WG12864F\)](#)⁴⁰⁸.
- [Solomon SSD1329 \(Ritdisplay RGS13128096\)](#)⁴¹⁰.
- [Himax HX8309 \(Ampire AM176220\)](#)⁴¹¹.



None of the above suits your project's needs? Tibbo can be contracted to develop a driver for another display you want to use. We offer reasonable development prices and ZNR (zero non-refundable) payment scheme, in which your initial payment for the driver development is gradually returned to you as you purchase Tibbo hardware.

Samsung S6B0108 (Winstar WG12864F)

Controller: Samsung S6B0108.

Panel: Winstar WG12864F and similar panels.

Type: LCD with blue backlight/white dots, monochrome (1 bit/pixel), vertical [pixel packing](#)³⁹³.

Locking: Supported, but the display image is not visible when the display is [locked](#)⁴⁰⁵. This causes a noticeable "glitch" on the display when the lock/unlock is performed.

Test hardware: TEV-LB0 test board. This board is a part of the EM1000-TEV development system. See Programmable Hardware Manual for details.

I/O mapping for WG12864F

This panel requires 6 I/O lines and an 8-bit data bus. Control lines are **RST**, **EN**, **CS1**, **CS2**, **D/I**, and **R/W**. Each control line can be connected to any I/O pin of your device, and each such I/O pin must be configured as an output (if your device requires explicit I/O line buffer configuration). The data bus can be connected to any 8-bit port. DO NOT configure this port for output.

The value of the [lcd.iomapping](#)⁴²² property must be set correctly for the display to work (see [Preparing the Display for Operation](#)³⁹⁵). For this particular display, the mapping string consists of 7 comma-separated decimal values:

1. Number of the I/O line connected to **RST**.
2. Number of the I/O line connected to **EN**.
3. Number of the I/O line connected to **CS1**.
4. Number of the I/O line connected to **CS2**.
5. Number of the I/O line connected to **D/I**.
6. Number of the I/O line connected to **R/W**.
7. Number of the I/O port connected to **DATA7-0**.

I/O line numbers are from the pl_io_num enum. Port number is from the pl_io_port_num enum. Line and port numbers are platform-specific. See the list of pl_io_num and pl_io_port_num constants in the platform specifications.

For the TEV-LB0 board, the lcd.iomapping property should be set to "44,46,40,41,43,42,4".

Code example -- TEV-LB0

This code will properly setup and enable this controller/panel (we assume that the testing is done using the TEV-LB0 board):

```
lcd.iomapping="44,46,40,41,43,42,4" 'RST,EN,CS1,CS2,D/I,R/W,DATA7-0

'configure control lines as outputs
io.num=PL_IO_NUM_46 'EN
io.enabled=YES
io.num=PL_IO_NUM_44 'RST
io.enabled=YES
io.num=PL_IO_NUM_40 'CS1
io.enabled=YES
io.num=PL_IO_NUM_41 'CS2
io.enabled=YES
io.num=PL_IO_NUM_42 'R/W
io.enabled=YES
io.num=PL_IO_NUM_43 'D/I
io.enabled=YES

'set resolution
lcd.width=128
lcd.height=64

'optionally set lcd.rotated here

lcd.enabled=YES 'done!

'turn on the backlight (strictly speaking, this is not related to the LCD
control, but we still show it here)
io.num=PL_IO_NUM_47
io.enabled=YES
io.state=HIGH

set_lcd_contrast(11) 'this is for external contrast control circuit
```

Contrast control

This is not a part of the panel itself, but we are still providing the code that will work on the TEV-LB0:

```

Sub set_lcd_contrast(level As Byte)
    'enable port, output data
    io.portnum=PL_IO_PORT_NUM_4
    io.portenabled=255
    io.portstate=level

    'generate strobe for the data register (on the LCD PCB)
    io.num=PL_IO_NUM_48
    io.enabled=YES
    io.state=HIGH
    io.state=LOW

    'disable port
    io.portenabled=0
End Sub

```

You can design your own contrast control circuit, of course.

Solomon SSD1329 (Ritdisplay RGS13128096)

Controller: Solomon SSD1329.

Panel: Ritdisplay RGS13128096 and similar panels.

Type: OLED, green, 16 levels (4 bits/pixel), horizontal [pixel packing](#)^[393].

Locking: Supported, but the display image is not visible when the display is [locked](#)^[405]. This causes a noticeable "glitch" on the display when the lock/unlock is performed.

Test hardware: TEV-LB1 test board. This board is a part of the EM1000-TEV development system. See Programmable Hardware Manual for details.

I/O mapping for RGS13128096

This panel requires 5 I/O lines and an 8-bit data bus. Control lines are **RES**, **D/C**, **R/W**, **E**, and **CS**. Each control line can be connected to any I/O pin of your device, and each such I/O pin must be configured as an output (if your device requires explicit I/O line buffer configuration). The data bus can be connected to any 8-bit port. DO NOT configure this port for output.

The controller also has **BS1** and **BS2** interface type selection pin. For proper operation, tie these pins to Vcc.

The value of the [lcd.iomapping](#)^[422] property must be set correctly for the display to work (see [Preparing the Display for Operation](#)^[395]). For this particular display, the mapping string consists of 6 comma-separated decimal values:

1. Number of the I/O line connected to **RES**.
2. Number of the I/O line connected to **D/C**.
3. Number of the I/O line connected to **R/W**.
4. Number of the I/O line connected to **E**.
5. Number of the I/O line connected to **CS**.
6. Number of the I/O port connected to **D7-0**.

I/O line numbers are from the `pl_io_num` enum. Port number is from the `pl_io_port_num` enum. Line and port numbers are platform-specific. See the list of `pl_io_num` and `pl_io_port_num` constants in your platform specifications.

For the TEV-LB0 board, `lcd.iomapping` should be set to "44,43,42,41,40,4".

Code example -- TEV-LB1

This code will properly setup and enable this controller/panel (we assume that the testing is done using the TEV-LB1 board):

```
lcd.iomapping="44,43,42,41,40,4" 'RST,D/C,R/W,E,CS,D7-0

'configure control lines as outputs
io.num=PL_IO_NUM_44 'RST
io.enabled=YES
io.num=PL_IO_NUM_43 'D/C
io.enabled=YES
io.num=PL_IO_NUM_42 'R/W
io.enabled=YES
io.num=PL_IO_NUM_41 'E
io.enabled=YES
io.num=PL_IO_NUM_40 'CS
io.enabled=YES

'set resolution
lcd.width=128
lcd.height=96

'optionally set lcd.rotated here

lcd.enabled=YES 'done!
```

Himax HX8309 (Ampire AM176220)

Controller: Himax HX8309.

Panel: Ampire AM176220 and similar panels.

Type: TFT, color, 16 bits/pixel.

Locking: Supported, display picture stays stable for at least 1 second after the display is [locked](#)^[41]. Unfortunately, there appears to be a hardware bug in the HX8309. This bug causes one of the horizontal lines of the panel (usually, the top or bottom line) to display random garbage while the display is locked. We have some ideas for workaround -- contact us if you encounter this problem.

Test hardware: TEV-LB2 test board. This board is a part of the EM1000-TEV development system. See Programmable Hardware Manual for details.

I/O mapping for AM176220

The panel requires 5 I/O lines and a 17-bit data bus, of which only bits DB7-0 are used (8-bit interface mode). Control lines are **RESET**, **RS**, **WR**, **RD**, and **CS**. Each control line can be connected to any I/O pin of your device, and each such I/O pin must be configured as an output (if your device requires explicit I/O line buffer configuration). The data bus can be connected to any 8-bit port. DO NOT configure this port for output.

The controller also has **IM0** and **IM3** interface type selection pin. For proper operation, tie IM0 to Vcc, IM3 - to the ground.

The value of the [`lcd.iomapping`](#)^[42] property must be set correctly for the display to work (see [Preparing the Display for Operation](#)^[35]). For this particular display, the mapping string consists of 6 comma-separated decimal values:

1. Number of the I/O line connected to **RESET**.

2. Number of the I/O line connected to **RS**.
3. Number of the I/O line connected to **WR**.
4. Number of the I/O line connected to **RD**.
5. Number of the I/O line connected to **CS**.
6. Number of the I/O port connected to **DB7-0**.

I/O line numbers are from the `pl_io_num` enum. Port number is from the `pl_io_port_num` enum. Line and port numbers are platform-specific. See the list of `pl_io_num` and `pl_io_port_num` constants in the platform specifications.

For the TEV-LB0 board, the `lcd.iomapping` should be set to "44,43,42,41,40,4".

Code example -- TEV-LB2

This code will properly setup and enable this controller/panel (we assume that the testing is done using the TEV-LB2 board):

```
lcd.iomapping="44,43,42,41,40,4" 'RESET,RS,WR,RD,CS,DB7-0

io.num=PL_IO_NUM_44 'RESET
io.enabled=YES
io.num=PL_IO_NUM_43 'RS
io.enabled=YES
io.num=PL_IO_NUM_42 'WR
io.enabled=YES
io.num=PL_IO_NUM_41 'RD
io.enabled=YES
io.num=PL_IO_NUM_40 'CS
io.enabled=YES

'set resolution
lcd.width=176
lcd.height=220

'optionally set lcd.rotated here

lcd.enabled=YES 'done!

'turn on the backlight (strictly speaking, this is not related to the LCD
control, but we still show it here)
io.num=PL_IO_NUM_47
io.enabled=YES
io.state=HIGH
```

Properties and Methods

The following classification groups properties and methods of the `lcd`. object by their logical function.

LCD panel characteristics:

- [lcd.paneltypes](#)^[424] [R/O Property]
- [lcd.pixelpacking](#)^[425] [R/O Property]
- [lcd.bitsperpixel](#)^[414] [R/O Property]
- [lcd.redbits](#)^[427] [R/O Property]
- [lcd.greenbits](#)^[420] [R/O Property]

- [Lcd.bluebits](#)⁴¹⁵ [R/O Property]

Preparing to work:

- [Lcd.iomapping](#)⁴²² [Property]
- [Lcd.width](#)⁴³² [Property]
- [Lcd.height](#)⁴²¹ [Property]
- [Lcd.inverted](#)⁴²² [Property]
- [Lcd.rotated](#)⁴²⁸ [Property]
- [Lcd.enabled](#)⁴¹⁶ [Property]

Graphical operations:

- [Lcd.setpixel](#)⁴²⁹ [Method]
- [Lcd.forecolor](#)⁴¹⁹ [Property]
- [Lcd.backcolor](#)⁴¹⁴ [Property]
- [Lcd.linewidth](#)⁴²³ [Property]
 - [Lcd.line](#)⁴²² [Method]
 - [Lcd.horline](#)⁴²¹ [Method]
 - [Lcd.verline](#)⁴³² [Method]
 - [Lcd.rectangle](#)⁴²⁷ [Method]
 - [Lcd.filledrectangle](#)⁴¹⁸ [Method]
 - [Lcd.fill](#)⁴¹⁷ [Method]
- [Lcd.setfont](#)⁴²⁸ [Method]
- [Lcd.fontheight](#)⁴¹⁸ [R/O Property]
- [Lcd.fontpixelpacking](#)⁴¹⁹ [R/O Property]
- [Lcd.textalignment](#)⁴²⁹ [Property]
- [Lcd.textorientation](#)⁴³⁰ [Property]
- [Lcd.texthorizontalspacing](#)⁴³⁰ [Property]
- [Lcd.textverticalspacing](#)⁴³¹ [Property]
 - [Lcd.print](#)⁴²⁵ [Method]
 - [Lcd.printaligned](#)⁴²⁶ [Method]
 - [Lcd.getprintwidth](#)⁴²⁰ [Method]
- [Lcd.bmp](#)⁴¹⁵ [Method]

Miscellaneous:

- [Lcd.error](#)⁴¹⁷ [R/O Property]
- [Lcd.lock](#)⁴²³ [Method]
- [Lcd.unlock](#)⁴³¹ [Method]
- [Lcd.lockcount](#)⁴²⁴ [R/O Property]

.Backcolor Property

Function:	Specifies current background color.
Type:	Word
Value Range:	0-65535. Default = 0.
See Also:	Understanding Controller Properties ^[393] , Working With Pixels and Colors ^[395] , lcd.forecolor ^[419] , lcd.linewidth ^[423]

Details

The background color is used when drawing filled rectangles ([lcd.filledrectangle](#)^[418]) and performing fills ([lcd.fill](#)^[417]).

Property value interpretation depends on the currently selected controller/panel. Selection is made through the Customize Platform dialog, accessible through the Project Settings dialog.

The property is of word type, but only the [lcd.bitsperpixel](#)^[414] lower bits of this value will be relevant. All higher bits will be ignored.

For monochrome and grayscale controllers/panels ([lcd.paneltype](#)^[424]= 0-PL_LCD_PANELTYPE_GRAYSCALE), this value will relate to the brightness of the pixel. For color panels/controllers ([lcd.paneltype](#)= 1- PL_LCD_PANELTYPE_COLOR) the value is composed of three fields -- one each for the red, green, and blue "channels". Check [lcd.redbits](#)^[427], [lcd.greenbits](#)^[420], and [lcd.bluebits](#)^[415] properties to see how the fields are combined into the color word.

.Bitsperpixel R/O Property

Function:	Returns the number of bits available for each pixel of the currently selected controller/panel.
Type:	Byte
Value Range:	Value depends on the currently selected controller/panel
See Also:	Understanding Controller Properties ^[393] , Working With Pixels and Colors ^[395]

Details

Controller/panel selection is made through the Customize Platform dialog, accessible through the Project Settings dialog.

For monochrome controllers/panels (see [lcd.paneltype](#)^[424]) the lcd.bitsperpixel will return 1, that is, the pixel can only be on or off. For grayscale panels, this value will be >1, which indicates that each pixel can be set to a number of brightness levels. For example, if the lcd.bitsperpixel=4, then each pixel's brightness can be adjusted in 16 steps.

For color panels, this property reflects the combined number of red, green, and blue bits available for each pixel (see [lcd.redbits](#)^[427], [lcd.greenbits](#)^[420], and [lcd.bluebits](#)^[415]).

The number of bits per pixel affects how [lcd.forecolor](#)^[419], [lcd.backcolor](#)^[414], and [lcd.setpixel](#)^[429] are interpreted. Also, the output produced by [lcd.bmp](#)^[415] depends

on this property.

.Bluebits R/O Property

Function:	A 16-bit value packing two 8-bit parameters: number of "blue" bits per pixel (high byte) and the position of the least significant blue bit within the color word (low byte).
Type:	Word
Value Range:	Value depends on the currently selected controller/panel.
See Also:	Understanding Controller Properties ³⁹³ , Working With Pixels and Colors ³⁹⁵

Details

The value of this property depends on the currently selected controller/panel. Selection is made through the Customize Platform dialog, accessible through the Project Settings dialog. This property is only relevant for color panels ([lcd.paneltype](#) ⁴²⁴= 1- PL_LCD_PANELTYPE_COLOR).

By taking the value of the high byte you can determine the number of the steps in which the brightness of the blue "channel" can be adjusted. For example, if the high byte is equal to 6, then there are 64 levels for blue.

This property also tells you the bit position and length of the blue field in color values used by [lcd.forecolor](#) ⁴¹⁹, [lcd.backcolor](#) ⁴¹⁴, and [lcd.setpixel](#) ⁴²⁹. If, for example, the [lcd.redbits](#) ⁴²⁷=&h0500, lcd.bluebits=&h0605, and [lcd.greenbits](#) ⁴²⁰=&h050B, then you can reconstruct the composition of the red, green, and blue bits in a word: bit 15 -> gggggbbbbbrrrr <- bit 0. In this example, the blue field is in the middle and occupies 6 bits (10-5).

.Bmp Method

Function:	Displays a portion of or full image stored in a BMP file.
Syntax:	lcd.bmp(offset as dword, x as word, y as word, x_offset as word, y_offset as word, maxwidth as word, maxheight as word) as ok_ng
Returns:	0- OK: Processed successfully. 1- NG: Unsupported or invalid file format.
See Also:	Displaying Images ⁴⁰⁴

Par	Description
t	

offs	Offset within the compiled binary of your application at which the BMP file is stored.
x	X coordinate of the top-left point of the image position on the screen. Value range is 0 to lcd.width ⁴³² -1.

y	Y coordinate of the top-left point of the image position on the screen. Value range is 0 to lcd.height ^[421] -1.
x_o ffse	Horizontal offset within the BMP file marking the top-left corner of the image portion to be displayed.
t	
y_o ffse	Vertical offset within the BMP file marking the top-left corner of the image portion to be displayed.
t	
ma xwi dth	Maximum width of the image portion to be displayed. Actual width of the output will be defined by the total width of the image and specified x_offset.
ma xhe ight	Maximum height of the image portion to be displayed. Actual height of the output will be defined by the total height of the image and specified y_offset.

Details

To obtain the offset, open the BMP file with [romfile.open](#)^[379], then read the offset of this file from the [romfile.offset](#)^[378] R/O property. Naturally, the BMP file must be present in your project for this to work (see how to [add](#)^[128] a file).

Note that only 2-, 16-, and 256-color modes are currently supported and the lcd.bmp will return 1- NG if you try to display any other type of BMP file. Compressed BMP files will be rejected too.

The method takes into account the type of the currently selected controller/panel (selection is made through the Customize Platform dialog, accessible through the Project Settings dialog). It will check the values of [lcd.paneltpe](#)^[424], [lcd.bitsperpixel](#)^[414], [lcd.redbits](#)^[427], [lcd.greenbits](#)^[420], and [lcd.bluebits](#)^[415] and produce the best output possible for the selected display.

X_offset, y_offset, maxwidth, and maxheight arguments allow you to display only a portion of the BMP image. This way it is possible to scroll around a large image that does not fit on the screen. Another use is to combine several separate images into a single file and display selected portions. This improves efficiency because [romfile.open](#)^[379] needs only be done once.

.Enabled Property

Function:	Specifies whether the display panel is enabled.
Type:	Enum (no_yes, byte)
Value Range:	0- NO: Disabled (default). 1- YES: Enabled.
See Also:	Preparing the Display for Operation ^[395]

Details

Several properties -- [lcd.iomapping](#)^[422], [lcd.width](#)^[432], [lcd.height](#)^[421], [lcd.inverted](#)^[422], [lcd.rotated](#)^[428] -- can only be changed when the display panel is disabled.

When you set this property to 1- YES, the controller of the panel is initialized and

enabled. This will only work if your display is properly connected, correct display type is selected in your project, `Lcd.iomapping` is set property, and necessary I/O lines are configured as outputs. The `Lcd.error`^[417] R/O property will indicate 1- YES if there was a problem enabling the display.

Setting the property to 0- NO disables the controller/panel.

.Error R/O Property

Function:	Indicates whether controller/panel I/O error has been detected.
Type:	Enum (no_yes, byte)
Value Range:	0- NO: No error detected. 1- YES: I/O error.
See Also:	Preparing the Display for Operation ^[395]

Details

The `Lcd` object will detect a malfunction (or absence) of the controller/panel that is expected to be connected. If the display is not properly connected, or the `Lcd` object is not set up property to work with this display, the `Lcd.error` will be set to 1- YES on attempt to enable the display (set `Lcd.enabled`^[416]= 1- YES).

.Fill Method

Function:	Paints the area with the "pen" color (Lcd.forecolor ^[419]).
Syntax:	<code>Lcd.fill(x as word,y as word, width as word, height as word)</code>
Returns:	---
See Also:	Lines, Rectangles, and Fills ^[396] , Working With Pixels and Colors ^[395] , Lcd.line ^[422] , Lcd.verline ^[432] , Lcd.horline ^[421] , Lcd.rectangle ^[427] , Lcd.filledrectangle ^[418]

Par	Description
t	
x	X coordinate of the top-left point of the area to be painted. Value range is 0 to Lcd.width ^[432] -1.
y	Y coordinate of the top-left point of the area to be painted. Value range is 0 to Lcd.height ^[421] -1.
wid	Width of the paint area in pixels.
th	
hei	Height of the paint area in pixels.
ght	

x	X coordinate of the top-left point of the area to be painted. Value range is 0 to [Lcd.width](#)^[432]-1.
y	Y coordinate of the top-left point of the area to be painted. Value range is 0 to [Lcd.height](#)^[421]-1.
wid	Width of the paint area in pixels.
th	
hei	Height of the paint area in pixels.
ght	

Details

The display panel must be enabled ([lcd.enabled](#)^[416]= 1- YES) for this method to work.

.Filledrectangle Method

Function: Draws a filled rectangle.

Syntax: **lcd.filledrectangle(x1 as word,y1 as word, x2 as word, y2 as word)**

Returns: ---

See Also: [Lines, Rectangles, and Fills](#)^[396], [Working With Pixels and Colors](#)^[395], [lcd.line](#)^[422], [lcd.verline](#)^[432], [lcd.horline](#)^[421], [lcd.rectangle](#)^[427], [lcd.fill](#)^[417]

Par t	Description
----------	-------------

- x1 X coordinate of the first point. Value range is 0 to [lcd.width](#)^[432]-1.
- y1 Y coordinate of the first point. Value range is 0 to [lcd.height](#)^[421]-1.
- x2 X coordinate of the second point. Value range is 0 to [lcd.width](#)^[432]-1.
- y2 Y coordinate of the second point. Value range is 0 to [lcd.height](#)^[421]-1.

Details

The border is drawn with the specified line width ([lcd.linewidth](#)^[423]) and "pen" color ([lcd.forecolor](#)^[419]). The rectangle is then filled using the background color ([lcd.backcolor](#)^[414]). Setting the lcd.linewidth to 0 will create a rectangle with no border -- basically, a filled area.

The display panel must be enabled ([lcd.enabled](#)^[416]= 1- YES) for this method to work.

.Fontheight R/O Property

Function: Returns the maximum height, in pixels, of characters in the currently selected font.

Type: Byte

Value Range: 0-255. **Default=** 0.

See Also: [Working With Text](#)^[395], [lcd.fontpixelpacking](#)^[419]

Details

This property will only return meaningful data after you select a font using the [lcd.setfont](#)^[428] method.

.Fontpixelpacking R/O Property

Function:	Indicates how pixels are packed into bytes in a currently selected font.
Type:	Enum (ver_hor, byte)
Value Range:	0- PL_VERTICAL: Several vertically adjacent pixels are packed into each byte of character bitmaps. 1- PL_HORIZONTAL: Several horizontally adjacent pixels are packed into each byte of character bitmaps.
See Also:	Working With Text ^[397] , Lcd.fontheight ^[418]

Details

Display controllers/panels can have vertical or horizontal pixel packing (see [Lcd.pixelpacking](#)^[425]). The speed at which you can output the text onto the screen is improved when the Lcd.pixelpacking and Lcd.fontpixelpacking have the same value, i.e. controller memory pixels and font encoding are "aligned". Our font files are typically available both in vertical and horizontal pixel packing. Pick the right file for your controller/panel.

This property will only return meaningful data after you select a font using the [Lcd.setfont](#)^[428] method.

.Forecolor Property

Function:	Specifies current "pen" (drawing) color.
Type:	Word
Value Range:	0-65535. Default = 65535 (&hFFFF).
See Also:	Understanding Controller Properties ^[393] , Working With Pixels and Colors ^[395] , Lcd.backcolor ^[424] , Lcd.linewidth ^[423]

Details

Pen color is used when drawing lines ([Lcd.line](#)^[422], [Lcd.verline](#)^[432], [Lcd.horline](#)^[421]) and rectangles ([Lcd.rectangle](#)^[427], [Lcd.filledrectangle](#)^[418]), as well as displaying text ([Lcd.print](#)^[425], [Lcd.printaligned](#)^[426]).

Property value interpretation depends on the currently selected controller/panel. Selection is made through the Customize Platform dialog, accessible through the Project Settings dialog.

The property is of word type, but only [Lcd.bitsPerPixel](#)^[414] lower bits of this value will be relevant. All higher bits will be ignored.

For monochrome and grayscale controllers/panels ([Lcd.paneltype](#)^[424]= 0-PL_LCD_PANELTYPE_GRAYSCALE), this value will relate to the brightness of the pixel. For color panels/controllers ([Lcd.paneltype](#)= 1- PL_LCD_PANELTYPE_COLOR) the value is composed of three fields -- one for the red, green, and blue "channels". Check [Lcd.redbits](#)^[427], [Lcd.greenbits](#)^[420], and [Lcd.bluebits](#)^[415] properties to see how the fields are combined into the color word.

.Getprintwidth Method

Function:	Returns the width, in pixels, of the text output that will be produced if the same line is actually printed with the lcd.print method.
Syntax:	lcd.getprintwidth(byref str as string) as word
Returns:	Total width of text output in pixels.
See Also:	Working With Text

Part	Description
------	-------------

str Text to estimate the output width for.

Details

This method does not produce any output on the display, it merely estimates the width of the text *if* it was to be printed. Lcd.print also returns the width of the text in pixels, but this data comes *after* the printing. Sometimes it is desirable to know the output width for the line of text before printing it, and this method allows you to do so.

The width calculation will be affected by the value of the [lcd.texthizontalspacing](#) property.

.Greenbits R/O Property

Function:	A 16-bit value packing two 8-bit parameters: number of "green" bits per pixel (high byte) and the position of the least significant green bit within the color word (low byte).
Type:	Word
Value Range:	Value depends on the currently selected controller/panel.
See Also:	Understanding Controller Properties , Working With Pixels and Colors

Details

The value of this property depends on the currently selected controller/panel. Selection is made through the Customize Platform dialog, accessible through the Project Settings dialog. This property is only relevant for color panels ([lcd.paneltypes](#)= 1- PL_LCD_PANELTYPE_COLOR).

By taking the value of the high byte you can determine the number of the steps in which the brightness of the green "channel" can be adjusted. For example, if the high byte is equal to 5, then there are 32 levels for green.

This property also tells you the bit position and length of the green field in values used by [lcd.forecolor](#), [lcd.backcolor](#), and [lcd.setpixel](#). If, for example, the [lcd.redbits](#)=&h0500, [lcd.bluebits](#)=&h0605, and [lcd.greenbits](#)=&h050B, then you can reconstruct the composition of the red, green, and blue bits in a word: bit

15 -> gggggbbbbbbrrrrr <- bit 0. In this example, the green field is the last field and occupies 5 bits (15-11).

.Height Property

Function:	Sets the vertical resolution of the display panel in pixels.
Type:	Word
Value Range:	Appropriate value depends on the panel. Default = 0.
See Also:	Preparing the Display for Operation ^[395] , lcd.width ^[432]

Details

Set this property according to the characteristics of your display panel.

This value is not set automatically when you select a certain controller because the capability of the controller may exceed the actual resolution of the panel, i.e. only "part" of the controller may be utilized.

This property can only be changed when the lcd is disabled ([lcd.enabled](#)^[416]= 0-NO).

.Horline Method

Function:	Draws a horizontal line.
Syntax:	lcd.horline(x1 as word,x2 as word,y as word)
Returns:	---
See Also:	Lines, Rectangles, and Fills ^[396] , lcd.rectangle ^[427] , lcd.filledrectangle ^[418] , lcd.fill ^[417]

Par	Description
t	
x1	X coordinate of the first point. Value range is 0 to lcd.width ^[432] -1.
x2	X coordinate of the second point. Value range is 0 to lcd.width ^[432] -1.
y	Y coordinates of the first and second points. Value range is 0 to lcd.height ^[421] -1.

Details

The line is drawn with the specified line width ([lcd.linewidth](#)^[423]) and "pen" color ([lcd.forecolor](#)^[419]). Drawing horizontal or vertical ([lcd.verline](#)^[432]) lines is more efficient than drawing generic lines ([lcd.line](#)^[422]), and should be used whenever possible.

The display panel must be enabled ([lcd.enabled](#)^[416]= 1- YES) for this method to work.

.Inverted Property

Function:	Specifies whether the image on the display panel has to be inverted.
Type:	Enum (no_yes, byte)
Value Range:	0- NO: Not inverted, higher memory value of the pixel corresponds to higher brightness (default). 1- YES: Inverted, higher memory value of the pixel corresponds to lower brightness.
See Also:	Preparing the Display for Operation [395]

Details

Set this property according to the characteristics of your display panel.

This value is not set automatically when you select a certain controller because the display characteristics cannot be detected automatically, as they depend on the panel and its backlight arrangement.

This property can only be changed when the display is disabled ([lcd.enabled](#) [416]=0- NO).

.Iomapping Property

Function:	Defines the list of I/O lines to interface with the currently selected controller/panel.
Type:	String
Value Range:	Value depends on the currently selected controller/panel. Default = "".
See Also:	Preparing the Display for Operation [395]

Details

Controller/panel selection is made through the Customize Platform dialog, accessible through the Project Settings dialog.

Different controllers/panels require a different set of interface lines, and even the number of lines depends on the hardware. This property should contain a comma-separated list of decimal numbers that indicate which I/O lines and ports are used to connect the controller/panel to your device. The meaning of each number in the list is controller- and panel-specific. See the [Supported Controllers](#) [408] section for details.

This property can only be changed when the display is disabled ([lcd.enabled](#) [416]=0- NO).

.Line Method

Function:	Draws a line.
Syntax:	lcd.line(x1 as word,y1 as word, x2 as word, y2 as word)

Returns:

See Also:

[Lines, Rectangles, and Fills](#)^[396], [lcd.rectangle](#)^[427], [lcd.filledrectangle](#)^[418], [lcd.fill](#)^[417]

Par	Description
-----	-------------

- x1 X coordinate of the first point. Value range is 0 to [lcd.width](#)^[432]-1.
- y1 Y coordinate of the first point. Value range is 0 to [lcd.height](#)^[421]-1.
- x2 X coordinate of the second point. Value range is 0 to [lcd.width](#)^[432]-1.
- y2 Y coordinate of the second point. Value range is 0 to [lcd.height](#)^[421]-1.

Details

The line is drawn with the specified line width ([lcd.linewidth](#)^[423]) and "pen" color ([lcd.forecolor](#)^[419]). Drawing horizontal ([lcd.horline](#)^[421]) or vertical ([lcd.verline](#)^[432]) lines is more efficient than drawing generic lines, and should be used whenever possible.

The display panel must be enabled ([lcd.enabled](#)^[416]= 1- YES) for this method to work.

.Linewidth Property**Function:**

Specifies current "pen" width in pixels.

Type:

Byte

Value Range:

1-255. **Default**= 1 (1 pixel).

See Also:

[Lines, Rectangles, and Fills](#)^[396], [lcd.forecolor](#)^[419], [lcd.backcolor](#)^[414]

Details

Pen width is used when drawing lines ([lcd.line](#)^[422], [lcd.verline](#)^[432], [lcd.horline](#)^[421]) and rectangles ([lcd.rectangle](#)^[427], [lcd.filledrectangle](#)^[418]).

.Lock Method**Function:**

Freezes display output (on controllers/panels that support this feature).

Syntax:

lcd.lock

Returns:

See Also:

Details

When the display is locked, you can make changes to the display data without showing these changes on the screen. You can then unlock the display (`lcd.unlock`) and show all the changes made at once. This usually greatly improves the display agility *perception* by the user (see [Improving Graphical Performance](#)^[405]).

When you execute this method for the first time, the display gets locked and the `lcd.lockcount`^[424] R/O property changes from 0 to 1. You can invoke `lcd.lock` again and again, and the `lcd.lockcount` will increase with each call to the `lcd.lock`. This allows you to nest locks/unlocks (again, see [Improving Graphical Performance](#)^[405]). Of course, the display is not locked "any harder" at `lcd.lockcount=2` compared to `lcd.lockcount=1`. The display is simply locked for all `lcd.lockcount` values other than 0.

Not all controllers/panels support this feature. See the [Supported Controllers/Panels](#)^[408] section for details on the display you are using. If your display does not support locking, executing `lcd.lock` will have no effect and `lcd.lockcount`^[424] will always stay at 0.

.Lockcount R/O Property

Function:	Indicates the current nesting level of the display lock.
Type:	Byte
Value Range:	0-255. Default = 0 (display unlocked).
See Also:	---

Details

Invoking `lcd.lock`^[423] increases the value of this property by 1. If 255 is reached, the value does not roll over to 0 and stays at 255. Invoking `lcd.unlock`^[431] decreases the value of this property by 1. When 0 is reached, the value does not roll over to 255 and stays at 0. The display is locked when `lcd.lockcount` is not at 0. Of course, the display does not get locked any "harder" with every increment of the `lcd.lockcount`.

When the display is locked, you can make changes to the display data without showing these changes on the screen. You can then unlock the display and show all the changes made at once. This usually greatly improves the display agility *perception* (see [Improving Graphical Performance](#)^[405]).

Not all controllers/panels support this feature. See the [Supported Controllers/Panels](#)^[408] section for details on the display you are using. If your display does not support locking, executing `lcd.lock` will have no effect and `lcd.lockcount`^[424] will always stay at 0.

.Paneltype R/O Property

Function:	Returns the type of the currently selected controller/panel.
Type:	Enum (<code>pl_lcd_paneltype</code> , byte)
Value Range:	0- <code>PL_LCD_PANELTYPE_GRAYSCALE</code> : This is a monochrome or grayscale panel/controller. 1- <code>PL_LCD_PANELTYPE_COLOR</code> : This is a color panel/controller.

See Also:

[Understanding Controller Properties](#)^[393], [Working With Pixels and Colors](#)^[395]

Details

Controller/panel selection is made through the Customize Platform dialog, accessible through the Project Settings dialog.

Monochrome panels/controllers only allow you to turn pixels on and off. Grayscale panels/controllers allow you to set the brightness of pixels in steps. The number of available steps is defined by the number of bits assigned to each pixel (see [lcd.bitsperpixel](#)^[414] property). Finally, color panels/controllers allow you to set the brightness separately for the red, green, and blue components of each pixel. [Lcd.redbits](#)^[427], [Lcd.greennbits](#)^[420], and [Lcd.bluebits](#)^[415] R/O properties will tell you how many bits there are for each color "channel".

Panel/controller type affects how [lcd.forecolor](#)^[419], [lcd.backcolor](#)^[414], and [lcd.setpixel](#)^[429] are interpreted. Also, the output produced by [lcd.bmp](#)^[415] is affected by this.

.Pixelpacking R/O Property

Function: Indicates how pixels are packed into controller memory for the currently selected controller/panel.

Type: Enum (ver_hor, byte)

Value Range: 0- PL_VERTICAL: Several vertically adjacent pixels are packed into each byte of controller memory.
1- PL_HORIZONTAL: Several horizontally adjacent pixels are packed into each byte of controller memory.

See Also: [Understanding Controller Properties](#)^[393]

Details

Controller/panel selection is made through the Customize Platform dialog, accessible through the Project Settings dialog.

This property is only relevant for controllers/panels whose [lcd.bitsperpixel](#)^[414] value is less than 8. In this case, 2, 4, or 8 pixels are packed into a single byte of controller memory.

This property is purely informational and largely has no influence over how you write your application. The only exception is related to [working with text](#)^[397]. [Fonts](#)^[400] can also have vertical or horizontal packing and the speed at which you can output the text onto the screen is improved when the [lcd.pixelpacking](#) and [lcd.fontpixelpacking](#)^[419] have the same value, i.e. controller memory pixels and font encoding are "aligned".

.Print Method

Function: Prints a line of text.

Syntax: **lcd.print(byref str as string,x as word,y as word) as word**

Returns: Total width of created output in pixels.

See Also: [Working With Text](#)^[397], [lcd.getprintwidth](#)^[420]

Part	Description
str	Text to print.
x	X coordinate of the top-left corner of the text output. Value range is 0 to lcd.width ^[432] -1.
y	Y coordinate of the top-left corner of the text output. Value range is 0 to lcd.height ^[421] -1.

Details

For this method to work, a font must first be selected with the [lcd.setfont](#)^[428] method. The [lcd.textorientation](#)^[430] and [lcd.texthorizontalspacing](#)^[430] properties affect how the text is printed.

This method always produces a single-line text output. Use [lcd.printaligned](#)^[426] if you want to print several lines of text at once.

The display panel must be enabled ([lcd.enabled](#)^[416]= 1- YES) for this method to work.

.Printaligned Method

Function:	Print texts, on several lines if necessary, within a specified rectangular area.
Syntax:	lcd.printaligned(byref str as string,x as word,y as word,width as word,height as word) as byte
Returns:	Total number of text lines produced.
See Also:	Working With Text ^[397]

Part	Description
str	Text to print. Inserting ` character will create a line break.
x	X coordinate of the top-left point of the print area. Value range is 0 to lcd.width ^[432] -1.
y	Y coordinate of the top-left point of the print area. Value range is 0 to lcd.height ^[421] -1.
wid	Width of the print area in pixels.
th	
hei	Height of the print area in pixels.
ght	

Details

For this method to work, a font must first be selected with the [lcd.setfont](#)^[428] method. The [lcd.textalignment](#)^[429], [lcd.textorientation](#)^[430], [lcd.texthorizontalspacing](#)^[430], and [lcd.textverticalspacing](#)^[431] properties will affect how the text is printed.

This method breaks the text into lines to stay within the specified rectangular output area. Whenever possible, text is split without breaking up the words. A word will be split if it is wider than the width of the print area. You can add arbitrary line breaks by inserting ` (ASCII code 96).

The display panel must be enabled ([lcd.enabled](#)^[416]= 1- YES) for this method to work.

.Rectangle Method

Function:	Draws an unfilled rectangle.
Syntax:	lcd.rectangle(x1 as word,y1 as word, x2 as word, y2 as word)
Returns:	---
See Also:	Lines, Rectangles, and Fills ^[396] , lcd.line ^[422] , lcd.verline ^[432] , lcd.horline ^[421] , lcd.filledrectangle ^[418] , lcd.fill ^[417]

Par	Description
t	

- x1 X coordinate of the first point. Value range is 0 to [lcd.width](#)^[432]-1.
- y1 Y coordinate of the first point. Value range is 0 to [lcd.height](#)^[421]-1.
- x2 X coordinate of the second point. Value range is 0 to [lcd.width](#)^[432]-1.
- y2 Y coordinate of the second point. Value range is 0 to [lcd.height](#)^[421]-1.

Details

The rectangle is drawn with the specified line width ([lcd.linewidth](#)^[423]) and "pen" color ([lcd.forecolor](#)^[419]).

The display panel must be enabled ([lcd.enabled](#)^[416]= 1- YES) for this method to work.

.Redbits R/O Property

Function:	A 16-bit value packing two 8-bit parameters: number of "red" bits per pixel (high byte) and the position of the least significant red bit within the color word (low byte).
Type:	Word
Value Range:	Value depends on the currently selected controller/panel.
See Also:	Understanding Controller Properties ^[393] , Working With Pixels and Colors ^[395]

Details

The value of this property depends on the currently selected controller/panel. Selection is made through the Customize Platform dialog, accessible through the Project Settings dialog. This property is only relevant for color panels ([lcd.paneltype](#)^[424]= 1- PL_LCD_PANELTYPE_COLOR).

By taking the value of the high byte you can determine the number of the steps in which the brightness of the red "channel" can be adjusted. For example, if the high byte is equal to 5, then there are 32 levels for red.

This property also tells you the bit position and length of the red field in values used by [lcd.forecolor](#)^[419], [lcd.backcolor](#)^[414], and [lcd.setpixel](#)^[429]. If, for example, the lcd.redbits=&h0500, [lcd.bluebits](#)^[415]=&h0605, and [lcd.greenbits](#)^[420]=&h050B, then you can reconstruct the composition of the red, green, and blue bits in a word: bit 15 -> gggggbbbbbbrrrrr <- bit 0. In this example, the red field is the first field and occupies 5 bits (4-0).

.Rotated Property

Function:	Specifies whether the image on the display panel is to be rotated 180 degrees.
Type:	Enum (no_yes, byte)
Value Range:	0- NO: Not rotated (default). 1- YES: Rotated 180 degrees.
See Also:	Preparing the Display for Operation ^[395]

Details

Set this property according to the orientation of the display panel in your device.

This property can only be changed when the display is disabled ([lcd.enabled](#)^[416]= 0- NO).

.Setfont Method

Function:	Selects a font to use for printing text.
Syntax:	lcd.setfont(offset as dword) as ok_ng
Returns:	0- OK: The font was found and the data appears to be valid. 1- NG: There is no valid font data at specified offset.
See Also:	Working with Text ^[397]

Par	Description
t	

offs Offset within the compiled binary of your application at which the font file is stored.

Details

A valid font file must be selected before you can use the [lcd.print](#)^[425], [lcd.printaligned](#)^[426], or [lcd.getprintwidth](#)^[420] methods. Naturally, the font file must be present in your project for this to work (see how to [add](#)^[128] a font file). To obtain correct offset, open the file using the [romfile.open](#)^[379] method, then read the offset of this file from the [romfile.offset](#)^[378] R/O property.

When the font file is successfully selected, the [lcd.fontheight](#)^[418] and [lcd.fontpixelpacking](#)^[419] R/O properties will be updated to reflect actual font parameters.

.Setpixel Method

Function:	Directly writes pixel data for a single pixel into the controller's memory.
Syntax:	lcd.setpixel(dt as word,x as word,y as word)
Returns:	---
See Also:	Working With Pixels and Colors ^[395] , Understanding Controller Properties ^[393]

Par t	Description
----------	-------------

dt	Pixel data to write.
x	X coordinate of the pixel. Value range is 0 to lcd.width ^[432] -1.
y	Y coordinate of the pixel. Value range is 0 to lcd.height ^[421] -1.

Details

Interpretation of the dt argument depends on the selected controller/panel. Selection is made through the Customize Platform dialog, accessible through the Project Settings dialog.

The dt argument is of word type, but only [lcd.bitsperpixel](#)^[414] lower bits of this value will be relevant. All higher bits will be ignored.

For monochrome and grayscale controllers/panels ([lcd.paneltype](#)^[424]= 0-PL_LCD_PANELTYPE_GRAYSCALE), the value of the dt argument sets the brightness of the pixel. For color panels/controllers ([lcd.paneltype](#)= 1-PL_LCD_PANELTYPE_COLOR) the value is composed of three fields -- one for the red, green, and blue "channels". Check [lcd.redbits](#)^[427], [lcd.greenbits](#)^[420], and [lcd.bluebits](#)^[415] properties to see how the fields are combined into the dt word.

The display panel must be enabled ([lcd.enabled](#)^[416]= 1- YES) for this method to work.

.Textalignment Property

Function:	Specifies the alignment for text output produced by the lcd.printaligned ^[426] method.
Type:	Enum (pl_lcd_text_alignment, byte)

Value Range:	0- PL_LCD_TEXT_ALIGNMENT_TOP_LEFT: Top, left (default). 1- PL_LCD_TEXT_ALIGNMENT_TOP_CENTER: Top, center. 2- PL_LCD_TEXT_ALIGNMENT_TOP_RIGHT: Top, right. 3- PL_LCD_TEXT_ALIGNMENT_MIDDLE_LEFT: Middle, left. 4- PL_LCD_TEXT_ALIGNMENT_MIDDLE_CENTER: Middle, center. 5- PL_LCD_TEXT_ALIGNMENT_MIDDLE_RIGHT: Middle, right. 6- PL_LCD_TEXT_ALIGNMENT_BOTTOM_LEFT: Bottom, left. 7- PL_LCD_TEXT_ALIGNMENT_BOTTOM_CENTER: Bottom, center. 8- PL_LCD_TEXT_ALIGNMENT_BOTTOM_RIGHT: Bottom, right.
---------------------	---

See Also: [Working With Text](#)^[397], [Lcd.textorientation](#)^[430],
[Lcd.texthorizontalspacing](#)^[430], [Lcd.textverticalspacing](#)^[431]

Details

[Lcd.printaligned](#)^[426] fits the text within a specified rectangular area. Lcd.textalignment defines how the text will be aligned within this area. The property has no bearing on the output produced by [Lcd.print](#)^[425].

.Texthorizontalspacing Property

Function:	Specifies the gap, in pixels, between characters of text output produced by the Lcd.print ^[425] and Lcd.printaligned ^[426] methods.
Type:	Byte
Value Range:	0- 255. Default = 1 (1 pixel).
See Also:	Working With Text ^[397] , Lcd.textAlignment ^[429] , Lcd.textorientation ^[430] , Lcd.textverticalspacing ^[431]

Details

.Textorientation Property

Function:	Specifies the print angle for text output produced by the Lcd.print ^[425] and Lcd.printaligned ^[426] methods.
Type:	Enum (pl_lcd_text_orientation, byte)

Value Range:	0- PL_LCD_TEXT_ORIENTATION_0: At 0 degrees (default). 1- PL_LCD_TEXT_ORIENTATION_90: At 90 degrees. 2- PL_LCD_TEXT_ORIENTATION_180: At 180 degrees. 3- PL_LCD_TEXT_ORIENTATION_270: At 270 degrees.
---------------------	--

See Also:	Working With Text ^[397] , lcd.textAlignment ^[429] , lcd.texthorizontalspacing ^[430] , lcd.textverticalspacing ^[431]
------------------	--

Details

.Textverticalspacing Property

Function:	Specifies the gap, in pixels, between the lines of text output produced by the lcd.printaligned ^[426] method.
Type:	Byte
Value Range:	0- 255. Default = 1 (1 pixel).
See Also:	Working With Text ^[397] , lcd.textAlignment ^[429] , lcd.textorientation ^[430] , lcd.texthorizontalspacing ^[431]

Details

The property has no bearing on the output produced by [lcd.print](#)^[425], because this method always creates a single-line output.

.Unlock Method

Function:	Unfreezes display output (on controllers/panels that support this feature).
Syntax:	lcd.unlock
Returns:	---
See Also:	---

Details

When the display is locked (see [lcd.lock](#)^[423]), you can make changes to the display data without showing these changes on the screen. You can then unlock the display with [lcd.unlock](#) and show all the changes made at once. This usually greatly improves the display agility perception by the user, even if your application is not working any faster (see [Improving Graphical Performance](#)^[405]).

Each time you execute this method on a previously locked display, the value of the [lcd.lockcount](#)^[424] R/O property decreases by 1. Once this value reaches 0, the display is unlocked and the user sees updated display data. The [lcd.lockcount](#)

allows you to nest locks/unlocks (again, see [Improving Graphical Performance](#)^[405]).

Not all controllers/panels support this feature. See the [Supported Controllers/Panels](#)^[408] section for details on the display you are using. If your display does not support locking, executing `lcd.lock` will have no effect and `lcd.lockcount`^[424] will always stay at 0.

.Verline Method

Function:	Draws a vertical line.
Syntax:	<code>lcd.verline(x as word,y1 as word,y2 as word)</code>
Returns:	---
See Also:	Lines, Rectangles, and Fills ^[396] , lcd.rectangle ^[427] , lcd.filledrectangle ^[418] , lcd.fill ^[417]

Part Description

x	X coordinates of the first and second points. Value range is 0 to lcd.width ^{[432]-1} .
y1	Y coordinate of the first point. Value range is 0 to lcd.height ^{[427]-1} .
y2	Y coordinate of the second point. Value range is 0 to lcd.height ^{[427]-1} .

Details

The line is drawn with the specified line width ([lcd.linewidth](#)^[423]) and "pen" color ([lcd.forecolor](#)^[419]). Drawing horizontal ([lcd.horline](#)^[421]) or vertical lines is more efficient than drawing generic lines ([lcd.line](#)^[422]) and should be used whenever possible.

The display panel must be enabled ([lcd.enabled](#)^[416]= 1- YES) for this method to work.

.Width Property

Function:	Sets the horizontal resolution of the display panel in pixels.
Type:	Word
Value Range:	Appropriate value depends on the panel. Default = 0.
See Also:	Preparing the Display for Operation ^[395] , lcd.height ^[427]

Details

Set this property according to the characteristics of your display panel.

The reason why this value is not set automatically when you select a certain controller is because the capability of the controller may exceed the actual resolution of the panel, i.e. only "part" of the controller may be utilized.

This property can only be changed when the display is disabled ([lcd.enabled](#)^[416]=

0- NO).

Fd Object



This is the flash disk (fd.) object, it allows you to utilize the unused portion of your flash memory for data storage. Your flash memory's primary use is to store the firmware of your device, as well as compiled Tibbo Basic application. The firmware/application typically occupy only a part of the flash memory. Available space will be referred to as the "data area" of the flash, as it can store your application's data.

There are two methods of working with the data area of the flash:

- With [file-based access](#)^[436], you create a formatted disk that stores files.
- With [direct sector access](#)^[448], you can write and read flash sectors directly, without burdening yourself with the file system.

Both methods can be used [concurrently](#)^[453] and complement each other whenever necessary.

Here is what the fd. object has to offer:

- Maintains up to 64 files located in a single root directory (subdirectories are not supported, but can be [emulated](#)^[441]).
- Flexible [file attributes](#)^[440] -- define and store any attributes you like.
- Methods to work with the [file directory](#)^[442].
- Ability to [open](#)^[443] several files at once.
- Methods to [write to and read from](#)^[444] the file, also cut the file size from the [beginning or end](#)^[445].
- Fast and flexible [search](#)^[446] to locate data within files. This also includes a **record-style** search!
- Automatic [sector leveling](#)^[453] in the "housekeeping" area of the file.
- Provisions for [safe recovery](#)^[454] from power failures.
- A method for firmware/application [self-upgrades](#)^[452].

Overview 3.13.1

In this section:

- [Sharing Flash Between Your Application and Data](#)^[434]
- [Fd. Object's Status Codes](#)^[435]
- [File-based Access](#)^[436]
- [Direct Sector Access](#)^[448]
- [File-based and Direct Sector Access Coexistence](#)^[453]
- [Prolonging Flash Memory Life](#)^[453]
- [Ensuring Disk Data Integrity](#)^[454]

Sharing Flash Between Your Application and Data

The `.fd` object uses the same physical flash chip that stores the firmware of your device, as well as your Tibbo Basic application. The flash memory consists of 264-byte sectors. It is customary to use 256 bytes of each sector to store actual data, and reserve the rest for "service" data, such as the checksum.

All flash sectors that are not occupied by the firmware/application are available for storing your data. We will refer to this area as a "data area". The size of the data area, in sectors, can be obtained through the `fd.availableflashspace`^[457] R/O property.

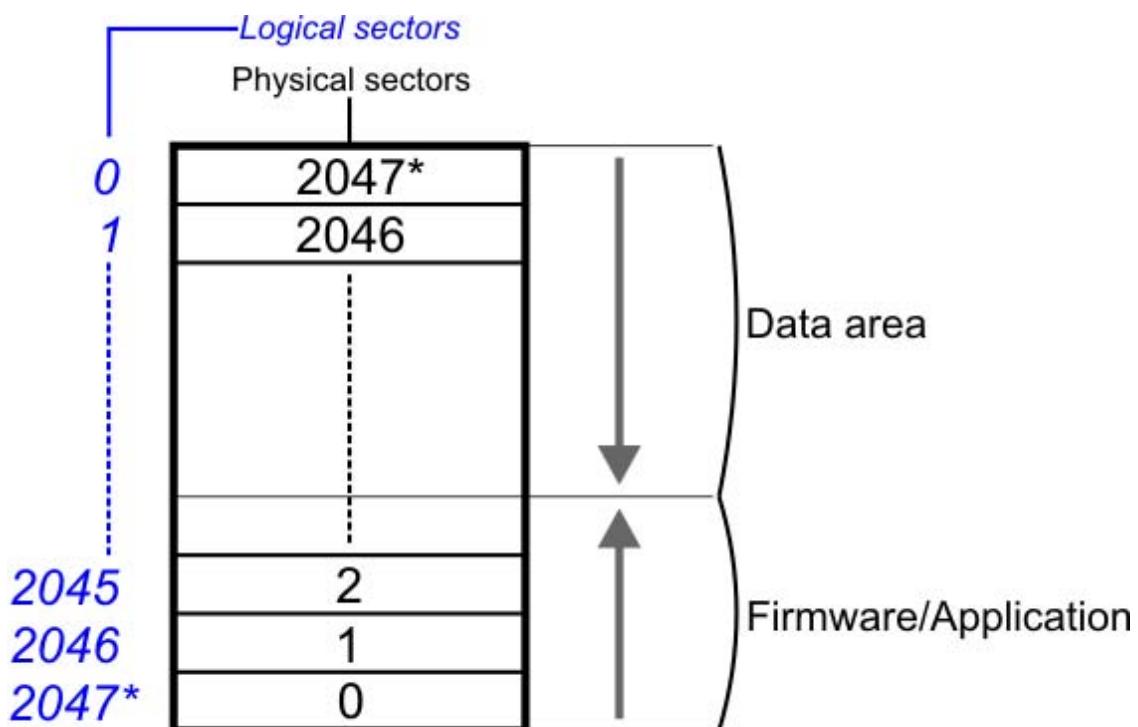
There are two ways in which you can use the data area of the flash chip.

The first way is to read and write flash sectors of the data area directly -- a so-called direct sector access^[448]. This method is simple and allows you to "do whatever you please" with the data area you have.

The second way is file-based^[436] -- you create a full-blown flash disk that maintains a simple file system. This introduces a certain overhead, since the file system needs a number of sectors for its own internal "housekeeping". At the same time, using files may be infinitely more convenient as many complex operations happen automatically and on the background, thus simplifying your application.

Both the direct sector access and file based access can be used at the same time -- your flash disk can occupy only a portion of the data area, and the rest of the space can be used for direct sector access.

The `fd` object refers to physical sectors of the data area in reverse. While the firmware and application are loaded into the flash memory starting from physical sector 0, the allocation for the data area starts from the topmost physical sector (see the drawing below). For convenience, related methods and properties of the `.fd` object refer to this topmost physical sector as (logical) sector #0, the sector before the topmost sector -- as #1, and so on.



* For 512K flash having 2048 x 256-byte sectors

This approach was chosen to minimize the influence of the changing size of your Tibbo Basic application on the data you keep in the flash memory. Typically, the size of your application keeps changing as you develop and debug it. At the same time, you often need to keep certain data in the data area permanently, and don't want to recreate this data every time you load new iteration of your application. If the beginning of the data area was right after the end of the firmware/application area, any change in the application size would move the boundary of the data area, corrupt your data, and force you to recreate the data again. If, on the other hand, your data area starts from the top of the flash IC and continues downwards, then the chance of the data area corruption will be a lot smaller.



When debugging your application, use only a portion of the data area and leave some of the data area unoccupied. This will create a gap of unused sectors between the data and the firmware/application areas of the flash. This way, your application will (mostly) be able to grow without corrupting the data area.

Fd. Object's Status Codes

Many things can go wrong when working with the flash memory. This is why most methods of the fd. object return a status code. Good Tibbo Basic application doesn't just assume that all will be well and always checks the status of method execution.

Listed below are possible status codes retuned by fd. object methods. Of course, not every code can be generated by every method:

- 0- PL_FD_STATUS_OK: Completed successfully.
- 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).
- 2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).
- 3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).
- 4- PL_FD_STATUS_INV_PARAM: Invalid argument have been provided for the invoked method.
- 5- PL_FD_STATUS_DUPLICATE_NAME: File with this name already exists.
- 6- PL_FD_STATUS_FILE_TABLE_FULL: Maximum number of files that can be stored on the disk has been reached, new file cannot be created.
- 7- PL_FD_STATUS_DATA_FULL: The disk is full, new data cannot be added.
- 8- PL_FD_STATUS_NOT_READY: The disk is not mounted.
- 9- PL_FD_STATUS_NOT_FOUND: File not found.
- 10- PL_FD_STATUS_NOT_OPENED: No file is currently opened "on" the current value of the [fd.filenum](#)^[463] property.
- 11- PL_FD_STATUS_ALREADY_OPENED: This file is already opened on some other

file number.

The status code generated by the most recently invoked method is always kept by the [fd.laststatus](#)^[472] R/O property. Some methods also return the status code directly:

```
If fd.create("File1.dat")<>PL_FD_STATUS_OK Then
    'some problem
End If
```

Other methods return data, so the only way to check the result of their execution is through the fd.laststatus:

```
s=fd.getdata(50) 'returns the data from the file, not the status code
If fd.laststatus<>PL_FD_STATUS_OK Then
    'some problem
End If
```

Some status codes are non-fatal, that is, they allow you to continue working. Some status codes indicate a serious problem after which your [flash disk](#)^[436] can't be used and must be [reformatted](#)^[436]. Such status codes are marked as "fatal" in the list above. If one of the fatal codes is generated, the flash disk is dismounted automatically and the [fd.ready](#)^[476] R/O property is set to 0- NO. The concept of disk mounting is explained in [Mounting the Flash Disk](#)^[439].

File-based Access

The following topics will explain how to work with the flash disk:

[Formatting the Flash Disk](#)^[436]

[Mounting the Disk](#)^[439]

[File Names and Attributes](#)^[440]

[Checking Disk Vitals](#)^[441]

[Creating, Deleting, and Renaming Files](#)^[441]

[Reading and Writing File Attributes](#)^[442]

[Walking Through File Directory](#)^[442]

[Opening Files](#)^[443]

[Writing To and Reading From Files](#)^[444]

[Removing Data From Files](#)^[445]

[Searching Files](#)^[446]

[Closing Files](#)^[448]

Formatting the Flash Disk

Before the flash disk can be used it must be formatted. Formatting allocates and initializes the "housekeeping" area of the disk. This consists of two boot sectors plus a certain number of sectors for the file record table (FRT) and the file allocation table (FAT). For the exact numbers see [Disk Area Allocation Details](#)^[437].

Formatting is performed using the [fd.format](#)^[467] method. This method accepts, as arguments, two parameters: total number of sectors occupied by the disk, and the maximum number of files that you wish to be able to store on the disk.

The total number of sectors cannot exceed the size of the [data area](#)^[434]. That is, the maximum "gross" disk size is [fd.availableflashspace](#)^[457] sectors. The flash disk needs several sectors for internal housekeeping reasons, so actual useful capacity of the disk will be less. [Checking Disk Vitals](#)^[441] topic explains how to find out current disk capacity, as well as get other useful info. There is also a minimum limit that will be accepted for the total disk size. The minimum exists because the number of sectors occupied by the disk must at least be enough for the "housekeeping" data.



When debugging your application, use less space for the disk than actually available -- this way the disk will not get corrupted each time your application size increases. When debugging is done, you can use maximum possible size, unless you want to leave some space for [sector backup](#)^[454].

The fd. object stores up to 64 files on the disk, in a single "root" directory. You can choose to have less files if you don't need so many. This will reduce the number of "housekeeping" sectors required for the disk. The economy is not dramatic, but you can still save some space. Each file record occupies 64 bytes, so one sector of the FRT table can store 4 file records. For this reason, the number of files you specify will always be adjusted to be the multiple of 4. For example:

```
'format a disk to occupy 3/4th of available space and store at least 10
files
If fd.format((fd.availableflashspace/4)*3,10)<>PL_FD_STATUS_OK Then
    'some problem...
End If
'actual maximum number of files will be 12 (adjusted up from 10 to be a
multiple of 4)
```

After the formatting the disk will be in the dismounted state and will need to be [mounted](#)^[439] before any disk-related activity can be successfully performed.

This topic provides details on the internal structure of the flash disk.

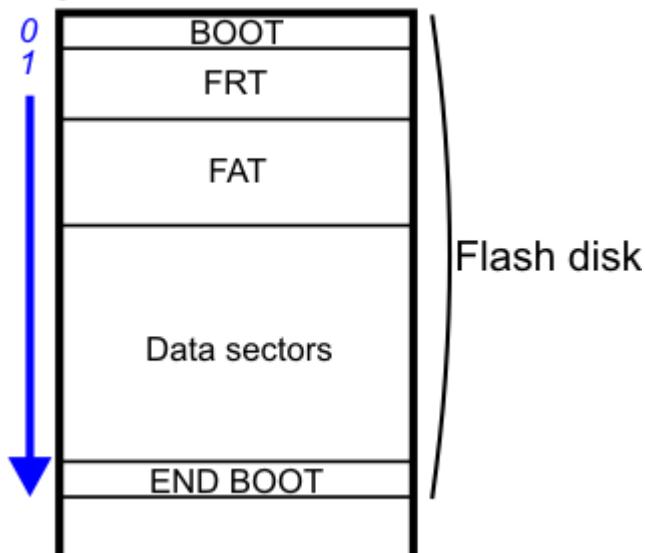
The flash disk has a number of areas. Each area includes one or more flash sectors. Each sector has a physical size of 264 bytes. Of those, only 256 bytes are used to store the data. Two more bytes are for the [checksum](#)^[450]. Six remaining bytes are mostly left unused, except in the FRT/FAT areas (see below), where they do serve a useful function.

The flash disk has the following areas:

- **BOOT** sector, always at logical sector #0 (so it is the topmost physical sector of the flash -- this was explained in [Sharing Flash Between Your Application and Data](#)^[434]). The boot sector contains the information about the sizes of other areas of the disk.
- File record table (**FRT**) area.
- File allocation table (**FAT**) area.
- **Data area** -- actual data sectors of files.

- **END BOOT** sector -- keeps the same data as the BOOT sector, but is located past the data area.

Logical sectors



FRT area

Each file record occupies 64 bytes. Therefore, each sector of the FRT area can fit 4 file records. Therefore, if you specify during formatting that you would like to have 15 different files, the [fd.format](#)^[467] method will round this up to 16 files. This will require 4 sectors to fit. Actual amount of allocated sectors is always *double* that. This is done for sector [leveling](#)^[453], which the fd. object takes care of internally. Also, the number of allocated sectors is never less than 8, again, for sector leveling reasons.

The maximum number of files stored by the .fd object is 64. Therefore, the maximum size of the FRT area is $(64/4)*2 = 32$ sectors. Hence, the size of the FRT area is always between 8 and 32 sectors depending on the maximum number of files you need to store on the disk.

FAT area

Each FAT sector consists of 128 FAT entries, 2 bytes per entry. Therefore, each FAT sector can fit the allocation data for 128 sectors from the data area of the disk. To improve sector leveling, the number of sectors allocated for the FAT area is, again, doubled against what's necessary. So, for every 128 sectors in the data area of the disk there are 2 sectors in the FAT area. At least 16 sectors are always allocated to FAT.

For example, supposing that the data area has 1100 sectors. Therefore, $1100/128 = 9$ FAT sectors are needed. The fd. object will allocate double this required minimum, so 18 FAT sectors will be prepared. Let's suppose now that we only have 500 data sectors. This requires 4 FAT sectors, 8 after we double this amount. This is less than the minimal 16 FAT sectors that are always provided, so the fd. object will still allocate 16 sectors.

Allocation and capacity calculation example

The fd.format method takes, as an input parameter, the total number of sectors

that you wish the disk to occupy. From that, and the supplied desired max number of files, the fd.format will work out the size of each area of the disk. Let's see this on a real example: Supposing, you call **fd.format(1300,41)** -- you want the disk to occupy 1300 sectors and store 41 file. So, what will be the size of the FRT, FAT, and data areas?

First, the fd.format deducts 2 sectors to account for the BOOT and the END BOOT sectors that must always be present. This leaves us with 1298 sectors.

Next, the fd.format determines the size of the FRT area: 41 is rounded to 44, then calculates that $(44/4)*2= 22$ sectors are needed (this exceeds the minimum of 8, so we do not need to correct this number). We have $998-22= 1276$ sectors left.

Now, how many FAT sectors we need? $1276/128= 10$. Actual amount will be doubled, so we need 20 sectors (exceeds the minimum of 16, so we do not need to correct this number). Therefore, we are left with $1276-20= 1256$ data sectors. This is what you will get from the [fd.capacity](#)⁴⁵⁸ R/O property (see [Checking Disk Vitals](#)⁴⁴¹).

One small caveat...

Now let us show you one trickier example. Say, you do **fd.format(1301,4)**. You can quickly work out that you have 1301 sector for the FAT and data areas combined. Now, how should this be divided?

If you had 1300 sectors, then there would be no problem. Of them, 1280 would go to the data area, and 20 will belong to the FAT. 10 "necessary" FAT sectors will together hold $10*128= 1280$ entries -- just right for the size of the data area. The number of FAT sectors is always doubled, so FAT will get 20 sectors. Everything fits perfectly!

Now, there is no good solution for 1301 sectors. If you give this extra one sector to the data area, then you will need two more sectors for the FAT area -- and you don't have those sectors. Allocating this extra sector to the FA is useless, because you are not increasing the number of data sectors, so you don't really need an extra FAT sector. Hence, the value of 1301 is actually not suitable to us. Same goes for 1302. 1303 is good again, because three extra sectors allow us to give one sector to the data area, and add two sectors to the FAT area.

Fd.format avoid falling into the trap by correcting downwards the total disk size you request if it turns out to be one of the problematic values. So, if you do **fd.format(1302,4)**, then the result would be as if you did **fd.format(1300,4)**. The [fd.totalsize](#)⁴⁸³ R/O property will show this to you (see [Checking Disk Vitals](#)⁴⁴¹).

Why is there the END BOOT sector?

The END BOOT sector exists for one purpose -- to detect whether your application has encroached on the flash disk. As was already [explained](#)⁴³⁴, the data area sectors are counted from the topmost physical sector of the flash memory down. Therefore, the END BOOT sector is the closest one to your application. Should the application become larger and take some of the space that was previously occupied by the flash disk, the END BOOT sector will be overwritten. The fd. object will detect this during [mounting](#)⁴³⁹ and return 3- PL_FD_STATUS_FORMAT_ERR status code.

Mounting the Flash Disk

The flash disk will not be accessible unless it is mounted using [fd.mount](#)⁴⁷⁴. It can only be mounted after the flash memory has been successfully formatted using [fd.format](#)⁴⁶⁷. The disk has to be mounted after every reboot of your device (if you need to access it, of course). After the disk is mounted successfully, the [fd.ready](#)

⁴⁷⁶ R/O property will read 1- YES.

During mounting, the fd. object accesses the [BOOT sectors](#)⁴³⁷ of the disk, read the size of all disk areas, and determines if the flash disk is healthy. This does not include detailed checks of each file or data sector of the disk, but is enough to "catch" gross problems, such as disk corruption due to an increase in the application size.

Once the disk has been mounted, you can [check its vitals](#)⁴⁴¹, [create and delete files](#)⁴⁴¹, and perform other disk-related operations.

There is no way to explicitly dismount the disk, nor it is necessary. The disk will be dismounted automatically if any fatal condition is detected when working with it. This condition will be reflected by the [fd.laststatus](#)⁴⁷² R/O property, while the [fd.ready](#)⁴⁷⁶ will become 0- NO. Not every error indicated by the fd.laststatus is fatal (see [Fd. Object's Status Codes](#)⁴³⁵ topic). The disk will also be dismounted if your application invokes [fd.format](#)⁴⁶⁷ or uses [fd.setsector](#)⁴⁸³ to write to a sector belonging to a mounted flash disk.

File Names and Attributes

Every file you create on the flash disk has a file name and attributes. Both share a single text string that can be up to 56 characters long. The attributes (if any) are separated from the file name by a space. In other words, everything up to the first space is the file name, and everything else -- attributes.

The file names are case-sensitive and can include any characters, except, obviously, the space. This includes "/" and "\". The flash disk does not support subdirectories, but it is possible to emulate them by including "/" or "\" characters in the file name. The "." character can be used to, so you can have any extension (s) you like. The attributes portion of the string may contain any characters whatsoever.

It is quite common for file systems to define attributes for the files. Typically, however, these attributes are preset. That is, you have a fixed list of things you can define about the file, such as the creating time, read-only flag, etc.

The fd. object uses a different approach. In a system that only runs a single application at any given time, it makes no sense to have, say, a fixed read-only flag. There is only one application running, after all. This application probably knows what files not to touch anyway! Our how about the date and time of the file creation? Does it make sense to keep this on a system without a real-time clock? Quite obviously, no.

For the above reasons, the fd. object allows you to store any attribute data and interpret it in any way you want. There is an attribute string and you can fill it with any data of your choosing and in accordance with the needs of your application. And yes, you can implement the read-only flag and record the creation date and time -- but only if you need to.

Here is an example where we [create a file](#)⁴⁴¹ and set its attributes too:

```
...
fd.create("File1.dat R 25-JUL-2008") ' <File1.dat> is the file name, <R
25-JUL-2008> -- attributes, it is up to our program how to interpret this!
```

Checking Disk Vitals

Once the disk is [mounted](#)⁴³⁹, you can check several important flash disk parameters:

- [Fd.capacity](#)⁴⁵⁸ will tell you the number of usable data sectors on the disk (reminder: each sector stores 256 bytes of data).
- [Fd.numservicesectors](#)⁴⁷⁵ will tell you how many sectors are used for internal "housekeeping" of the flash disk.
- [Fd.totalsize](#)⁴⁸³ will indicate how many sectors the disk occupies in flash memory. Fd.totalsize=fd.capacity+fd.numservicesectors.
- [Fd.getfreespace](#)⁴⁷⁰ method will return the number of free data sectors on the disk.

Creating, Deleting, and Renaming Files

You can't really do anything useful with the flash disk unless you create at least one file. The [fd.create](#)⁴⁶² method is used for this. The string you supply as an argument must include a file name and may also contain the [attributes](#)⁴⁴⁰. Some examples:

```
If fd.create("File1.dat R 25-JUL-2008")<>PL_FD_STATUS_OK Then '
<File1.dat> is the file name, <R 25-JUL-2008> -- attributes.
'some problem
End If
If fd.create("    File2")<>PL_FD_STATUS_OK Then 'no attributes defined for
this file. Notice leading spaces -- they will be removed.
'some problem
End If
If fd.create("Database/users.dat")<>PL_FD_STATUS_OK Then 'and here we
emulate a directory
'some problem
End If
If fd.create("file 3")<>PL_FD_STATUS_OK Then 'if the idea was to create a
<file 3> file, then this won't work! "3" will be interpreted as
attributes!
'some problem
End If
```

Naturally, each file on the flash disk must have a unique name, or the 5-PL_FD_STATUS_DUPLICATE_NAME error will be generated. Every existing file always has at least one data sector allocated to it. This is how the 7-PL_FD_STATUS_DATA_FULL error may be generated when you are creating a new file. Finally, the total number of files stored on the flash disk is limited to what you defined when [formatting](#)⁴³⁶ your disk. This maximum can be checked through the [fd.maxstoredfiles](#)⁴⁷⁴ R/O property. Try to exceed this number and you will get the 6- PL_FD_STATUS_FILE_TABLE_FULL error code.

To delete a file, use the [fd.delete](#)⁴⁶³ method:

```
If fd.delete("File1.dat")<>PL_FD_STATUS_OK Then
    'some problem
End If
If fd.delete("File2 abc")<>PL_FD_STATUS_OK Then 'the "abc" part will be
    ignored -- everything after the space is NOT a part of the file name
    'some problem
End If
If fd.delete("Database/users.dat")<>PL_FD_STATUS_OK Then 'leading
    spaces will be ignored
    'some problem
End If
```

The file you are deleting must exist, of course, or you will get the 9-PL_FD_STATUS_NOT_FOUND error. It is OK to delete a file which is currently opened^[443].

You can also rename a file using the [fd.rename](#)^[477] method.

Reading and Writing File Attributes

You can set initial file [attributes](#)^[440] right when [creating](#)^[441] a new file. To read existing file attributes, use the [fd.getattributes](#)^[467] method:

```
'this example ignores potential error conditions
Dim s As String
...
s=fd.getattributes("File1.dat") 'get the attributes for this file
If instr(1,s,"R",1)=0 Then 'delete the file only if there is no 'R' in the
    attributes
    fd.delete("File1.dat")
End If
```

You can set new (change) attributes with the [fd.setattributes](#)^[478] method:

```
If fd.setattributes("File1.dat", "D")<>PL_FD_STATUS_OK Then 'the attribute
    string will now consist of a single 'D'
    'some problem
End If
```

Remember that the file name and the attributes share the same 56-byte string in the file record table of the flash disk. Therefore, the maximum length of the attributes is limited to 56 -- length_of_the_file_name-. This "-1" accounts for the space that separates the file name from the attributes.

Attempting to perform the [fd.getattributes](#) or [fd.setattributes](#) on a file that does not exist will generate the 9- PL_FD_STATUS_NOT_FOUND error.

Walking Through File Directory

At times it is necessary to get the list of all files stored on the flash disk. Most operating systems provide this feature in the form of a DIR command. The fd. object offers two methods -- [fd.resetdirpointer](#)^[478] and [fd.getnextdirmember](#)^[470] -- that allow you to get the names of all stored files one by one.

An imaginary "directory pointer" walks through the file record table (FRT) of the flash disk. The fd.resetdirpointer method resets the directory pointer to 0, i.e. to the very first directory record. The fd.getnextdirmember returns the next file name and advances the pointer. Only file names are returned, [attributes](#)⁴⁴² are excluded.

Calling the fd.getnextdirmember repeatedly will get you all the file names. When there are no more names left to go through, the method will return an empty string. You can use this to end your directory walk:

```
'walk through the file directory -- method #1
Dim s As String

fd.resetdirpointer
Do
    s=fd.getnextdirmember
    If fd.laststatus<>PL_FD_STATUS_OK Then
        'some problem
    End If
    ... 'process the file name in s
Loop While s<>"" 'we exit on empty string
```

Alternatively, you can use a for-next cycle, since the number of currently stored files can be checked through the [fd.getnumfiles](#)⁴⁷¹ method:

```
'walk through the file directory -- method #2
Dim s As String
Dim f As Byte

fd.resetdirpointer
For f=1 To fd.getnumfiles
    s=fd.getnextdirmember
    If fd.laststatus<>PL_FD_STATUS_OK Then
        'some problem
    End If
    ... 'process the file name in s
Next f
```

Opening Files

You must first open a file in order to work with its data. The [fd.open](#)⁴⁷⁵ method opens a file with a specified name and "on" a file number currently selected by the [fd.filenum](#)⁴⁶³ property. All operations related to the file data are then performed by referring to the file number, not the file name. These operations include [writing to and reading from files](#)⁴⁴⁴, [removing data from files](#)⁴⁴⁵, [searching files](#)⁴⁴⁶, and [closing files](#)⁴⁴⁸.

The concept of file numbers is not new -- other operating systems, too, assign a number to the file when the file is opened. In Tibbo Basic, you select the number you want to open the file on yourself. You do this by selecting a desired value for the fd.filenum property:

```
'will open two files 'on' numbers 3 and 5

fd.filenum=3
If fd.open("File1")<>PL_FD_STATUS_OK Then
    'some problem
End If
fd.filenum=5
If fd.open("TrestFile")<>PL_FD_STATUS_OK Then
    'some problem
End If
```

Whenever you want to work with one of the currently opened files, just set the fd.filenum to the number on which this file was opened (naturally, you need to somehow remember this number). And how many files can be opened concurrently? The [fd.maxopenedfiles](#)^[473] R/O property will tell you that. This value is platform-dependent. Your fd.filenum value can move between 0 and fd.maxopenedfiles-1.

When the file is opened "on" a certain file number, the [fd.fileopened](#)^[464] R/O property returns 1- YES when this file number is selected in the fd.filenum.

Several additional notes:

Any leading spaces in the file name you supply for fd.open are removed. After that, only the part up to the first space is processed -- the rest of the string is ignored. The following three code lines all open the same file:

```
fd.open("File1")
fd.open("      File1") 'leading spaces will be removed
fd.open("File1 some more stuff") 'everything after the first space will be
ignored too
```

Naturally, the file with the specified name must exist, or you get the 9-PL_FD_STATUS_NOT_FOUND error. You may not open the same file "on" two different file numbers -- this will generate the 11-PL_FD_STATUS_ALREADY_OPENED error. You may reopen the same file "on" the same file number, but this can lead to the loss of (some) changes made to the file prior to re-opening. To avoid this, [close](#)^[468] the file or use the [fd.flush](#)^[466] method before opening it again. Note that the fd.flush method does not depend on the current fd.filenum value and works globally on any most recently changed file.

Writing To and Reading From Files

Writing to and reading from files are two most important file operations. Having a flash disk would be pointless without them. Writing and reading is done using two methods -- [fd.setdata](#)^[480] and [fd.getdata](#)^[469] respectively. These work on a currently selected file, and the selection is made through the [fd.filenum](#)^[463] property.

Both reading and writing operations are always performed from the *current pointer position*. This position can be checked through the [fd.pointer](#)^[476] R/O property and changed through the [fd.setpointer](#)^[482] method. Executing the fd.setdata or fd.getdata moves the pointer forward by the number of file positions written or read.

The pointer always points at the position (offset) in the file from which the next reading or writing will be done. File offsets are counted from one, not zero. The very first byte in the file is at offset one, the next byte -- at offset two, and so on. The very last byte in the file is at offset equal to the size of the file, which is indicated by the [fd.filesize](#)^[464] R/O property. The maximum pointer value, however,

is `fd.filesize+1`! When the pointer is at maximum, it effectively points at the file position that doesn't exist yet. So, writing to the file at this position will append new data to the end of the file. Writing to the file when the pointer is not at maximum will overwrite existing data.

When the file is [opened](#)^[443], the pointer is set to 1 if the file has any data in it (`fd.filesize<>0`), or 0 if the file is empty (`fd.filesize=0`). It is not possible to set the pointer to zero if the file is not empty. The pointer will be set to zero automatically if the file becomes empty.

In the following example, we append the data to the end of the file, then read the data from the beginning of the file:

```
Dim s As String

'open a file 'on' file number 3
fd.filenum=3
fd.open("SomeFile")
fd.setpointer(fd.filesize+1) 'works always -- no matter whether the file
is empty or not
fd.setdata("Append this to the file")
fd.setpointer(1) 'go back to the beginning of the file
s=fd.getdata(10) 'read 10 bytes from the beginning of the file
```

As you append new data to the file, the file will grow larger. New data sectors will be allocated and added to the file automatically as needed. You will get the 7-`PL_FD_STATUS_DATA_FULL` error if the disk runs out of free sectors.

The `fd.` object uses a RAM buffer as an intermediary storage for the sector data. When you access a certain data sector, the contents of this sector are loaded into the RAM buffer automatically. When you change the data in the sector, it is the data in the RAM buffer that gets changed. The changed contents of the RAM buffer will not be saved back to the flash memory until the contents of *another* sector must be loaded into the buffer. So, for example, if you do this...

```
fd.setdata("Write this to a file")
```

... and then leave the disk alone, then the new data may stay in the RAM buffer indefinitely. It may get lost -- for example, if you reboot the device. To prevent this, [close](#)^[448] the file or use the [fd.flush](#)^[466] method -- either one will cause the changed data in the RAM buffer to be saved back to the flash memory. Note that the `fd.flush` method does not depend on the current `fd.filenum` value and works globally on any most recently changed file.

Finally, do take note of the [fd.sector](#)^[478] R/O property. This property reflects the number of the data sectors corresponding to the current position of the pointer. This information can be useful, for example, to [backup](#)^[454] the data sector before making changes to the file.

Removing Data From Files

Using the [fd.setdata](#)^[480] method can only increase the size of your file. To reduce the file size, i.e. remove some data from it, use one of the two following methods. Both methods work on a currently selected file, and the selection is made through the [fd.filenum](#)^[463] property.

The [fd.setfilesize](#)^[481] method cuts the end portion of your file and preserves a

specified amount of bytes in the beginning of the file:

```
'open a file 'on' file number 4
fd.filenum=4
fd.open("SomeFile")
fd.setfilesize(fd.filesize/2) 'cut the file size in half'
```

The `fd.setfilesize` can't be used to enlarge your file, only to make it smaller. Data sectors previously allocated to the file will be "released" (marked unused) if they become unnecessary due to the reduction in the file size. The first data sector of the file, however, will always remain allocated, even when the file size is set to 0.

The size of the file, indicated by the `fd.filesize`^[464] R/O property, will be corrected downwards to reflect the amount of data left in the file.

The `pointer position`^[444] will be affected by this method. If the file becomes empty, the pointer will be set to 0. If the new file size is not zero, but the new size makes current pointer position invalid (that is, `fd.pointer > fd.filesize+1`) then the pointer will be set to `fd.filesize+1`.

The second method -- `fd.cutfromtop`^[465] -- removes a specified number of *sectors* (not bytes) from the beginning of the file and leaves the end portion of the file intact:

```
'open a file 'on' file number 2
fd.filenum=2
fd.open("SomeFile")
fd.cutfromtop(3) 'remove three front sectors from the file (that is,
remove up to 3*256= 768 bytes of data)'
```

The size of the file will be corrected downwards in accordance with the amount of removed data. For example, performing `fd.cutfromtop(2)` on a file occupying 3 data sectors will reduce its size by 512 bytes (amount of data in 2 sectors removed). Performing `fd.cutfromtop(3)` will set the file size to 0.

The pointer position is always reset as a result of this method execution. If the new file size is 0, then the pointer will be set to 0 as well. If the file size is not 0, then the pointer will be set to 1.

Searching Files

The `fd.find`^[464] method allows you to quickly search through the file from a specified starting position, either in forward or back direction, and locate the Nth instance of a search substring. The method also allows you to specify the search "increment" (step). The search is performed on a currently selected file, and the selection is made through the `fd.filenum`^[463] property.

The search returns the position within the file, counting from 1, where the target occurrence of the substring has been encountered, or 0 if the target occurrence of the substring was not found.

The increment parameter is very important as it allows you to perform two fundamentally different classes of search.

Full text search

Set the increment to 1 and you will search through the entire contents of the file:

```

Dim dw As dword

'try to find the 2nd occurrence of 'ABC', search forward starting at the
beginning of the file
dw=fd.find(1,"ABC",2,FORWARD,1)
If fd.laststatus<>PL_FD_STATUS_OK Then
    'some disk-related error
Else

If dw<>0 Then
    'found! process this...
Else
    'not found...
End If

```

Since the search substring in the above example -- "ABC" -- does not have repeating fragments, you can actually set the search increment to 3 (the length of the substring). This will improve the search speed:

```
fd.find(1,"ABC",2,FORWARD,3) 'no repeating fragments in the substring, use its length as increment
```

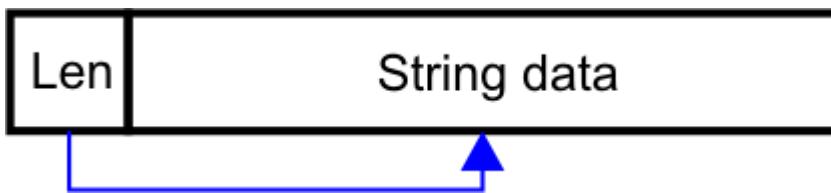
Record-style search

If the file in question is a data table consisting of individual records, then you can arrange for a very efficient search for the record with desired field value. All you need to achieve this is to have record fields occupy fixed offsets within records. For example, supposing you have the data table consisting of records with the following structure:

Field name	Offset in bytes*	Length in bytes
Category	+0	1
ID-code	+1	11
Last name	+12	21
First name	+43	21

* with respect to the beginning of the record

Each record of this data table occupies 54 bytes, so this will be our step. Three fields -- "ID-code", "last name", and first name" are strings which, of course, can have a variable length. To facilitate the use of the fd.find method, each field must reside at a fixed offset relative to the beginning of the record. That is, even if the "ID-code" for a particular record is shorter than maximum possible field length, the "Last name" field will still be at offset +12. To reflect actual length of the field data, each field of string type starts with a byte that denotes the length of the string, followed by the string data itself:



Now let's suppose you want to find a record whose "last name" is "Smith". For this we search starting from file offset 21, which is the offset of the "Last name" field within the record (this assumes that the first record starts right from the beginning of the file, which is usually the case). The search step will be 54 -- the size of the record:

```

Dim dw As dword

'try to find the record with the "Last name" set to "Smith"
dw=fd.find(21,chr(5)+"Smith",1,FORWARD,54) 'notice how we supply the
string length
If fd.laststatus<>PL_FD_STATUS_OK Then
    'some disk-related error
Else

    If dw<>0 Then
        'found- convert into the record number and process...
        dw=dw/54
    Else
        'not found...
    End If

```

You can also search back, but remember that this is less efficient (takes ~ 50% longer) compared to forward searches.

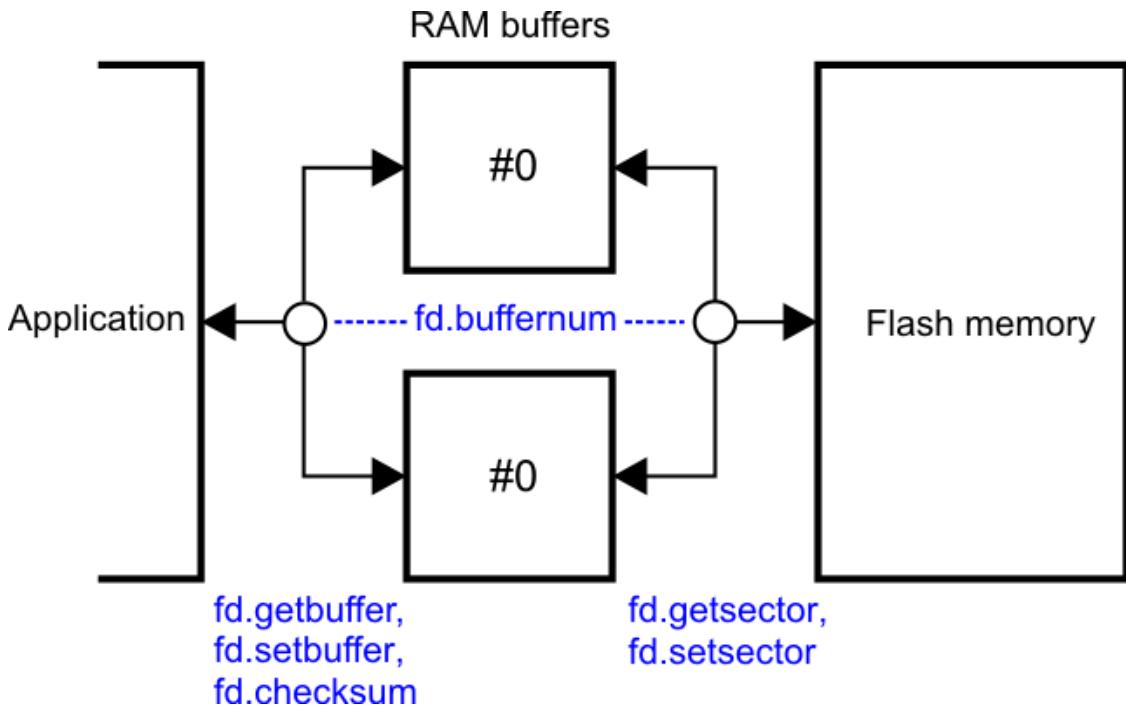
Closing Files

Files are closed using the [fd.close](#)^[460] method. The method is performed on a currently selected file (selection is made through the [fd.filenum](#)^[463] property).

Proper file closing is only required if you made changes to the file. If you only read data from the file you don't have to bother closing it -- you can [open](#)^[443] another file right "on" the same file number. Doing so on a file that was changed may cause some recent changes to be lost. To prevent this, use the [fd.close](#) method or [fd.flush](#)^[466] method. Note that [fd.flush](#) does not depend on the current [fd.filenum](#) value and works globally on any most recently changed file.

Direct Sector Access

Direct sector access allows you to work with physical sectors of the flash chip directly, without the need to create and manage any files. You work with sectors through two identical 264-byte RAM buffers numbered #0 and #1 (each flash sector contains 264 bytes of data). The [fd.buffernum](#)^[458] property selects one of the buffers as the source/destination for the data (see the drawing below).



Reading data from a sector is a two-step process. First, you use [fd.getsector](#)^[472] to load all 264 bytes from desired sector into the currently selected RAM buffer. Next, you use [fd.getbuffer](#)^[468] to read the data from the selected buffer. This method allows you to read up to 255 bytes beginning from any offset in the buffer (offsets are counted from 0):

```
Dim s As String
...
'we want bytes 20-29 from logical sector #3
fd.buffernum=0 'select RAM buffer #0
If fd.getsector(3)<>PL_FD_STATUS_OK Then
    'flash failure
End If
s=fd.getbuffer(20,10) 'now s contains desired data
```

Since the sector (and RAM buffer) size exceeds 255 bytes (maximum length of string variables), you can't actually read the whole sector contents in one portion. At least two `fd.getbuffer` reads are necessary for that.

To modify the data in the selected RAM buffer, use the [fd.setbuffer](#)^[479] method. The `fd.setbuffer` allows you to write new data at any offset of the selected RAM buffer. To store the contents of the RAM buffer back to the flash memory, use the [fd.setsector](#)^[483] method:

```

Dim s As String
...
'modify first 3 bytes of logical sector #5
If fd.getsector(3)<>PL_FD_STATUS_OK Then
    'flash failure
End If
fd.setbuffer("ABC",0) 'write "ABC" to the buffer at offset 0
If fd.setsector(3)<>PL_FD_STATUS_OK Then
    'flash failure
End If

```

Since there are two identical RAM buffers, you can load the contents of two different sectors and work with the data concurrently, by switching between the buffers.

As covered under [Sharing Flash Between Your Application and Data](#)^[434], logical sector numbers for fd.getsector and fd.setsector are not actual physical sector numbers of the flash IC. Logical sector #0 corresponds to the topmost physical sector of the flash IC, so logical numbering is "in reverse".

You can only write to sectors that reside within the data area of your flash chip (that is, logical sector numbers from 0 to [fd.availableflashspace](#)^{[457]-1}). This is done to prevent your application from inadvertently damaging its own code or the firmware. Trespassing the data area boundary will result in the 1-PL_FD_STATUS_FAIL [status code](#)^[435]. If you want to alter the data in the firmware/application area, see [Upgrading the Firmware/Application](#)^[452] topic.

If you are using direct sector access and [file-based access](#)^[436] at the same time, be sure to read about ensuring their proper [coexistence](#)^[453].

Using Checksums

When dealing with the flash memory, it is very often desirable to make sure that the data stored in flash sectors is not corrupted. One way to do so is by calculating the checksum on the sector data and storing this checksum together with the data. Each 264-byte sector of the flash memory conveniently has 8 extra bytes of memory on top of the "regular" 256 bytes existing to store actual data.

The fd. object uses first two of the 8 spare storage locations to keep the checksum of the 256-byte data block residing in the same sector (see the drawing below). The checksum is a 16-bit value. When the checksum is correct, a 16-bit sum of the 256-byte data block plus the checksum itself should be zero. Of course, this is an arbitrary choice -- there is no special reason why the .fd object does it this way, and not in any other way. We just decided to do it like that, and that was it!

	0	12	34	5	E	F
010	H	L	H	L	H	L
020	H	L	H	L	H	L
030	H	L	H	L	H	L
0F0	H	L	H	L	H	L
100	H	L				

Checksum

16-bit values for these calculations are little-endian. That is, offset 0 of the sector is presumed to be the high byte of the first 16-bit value, offset 1 -- low byte of the first 16-bit value, offset 2 -- high byte of the second 16-bit value, and so on.

When you are using the [file-based access](#)^[436], the fd. object automatically calculates and/or verifies the checksum on all sectors it accesses. Should any sector turn out to contain an invalid checksum, the 2-PL_FD_STATUS_CHECKSUM_ERR [status code](#)^[435] is generated.

Direct sector access is more primitive and it is your responsibility to maintain and verify the integrity of data you store in the flash memory. To aid you in this, the [fd.checksum](#)^[459] method that can be used to both verify and calculate the checksum of the sector's data:

```
'Load sector #10, verify its checksum, alter some data, recalculate the
checksum, and save new data
Dim i As word
...
'load the sector
If fd.getsector(10)<>PL_FD_STATUS_OK Then
    'flash failure
End If

'verify the checksum
If fd.checksum(PL_FD_CSUM_MODE_VERIFY,i)<>OK Then
    'checksum error detected
End If

fd.setbuffer("ABC",20) 'alter data at offset 20

fd.checksum(PL_FD_CSUM_MODE_CALCULATE,i) 'recalculate the checksum

'save back
If fd.setsector(10)<>PL_FD_STATUS_OK Then
    'flash failure
End If
```

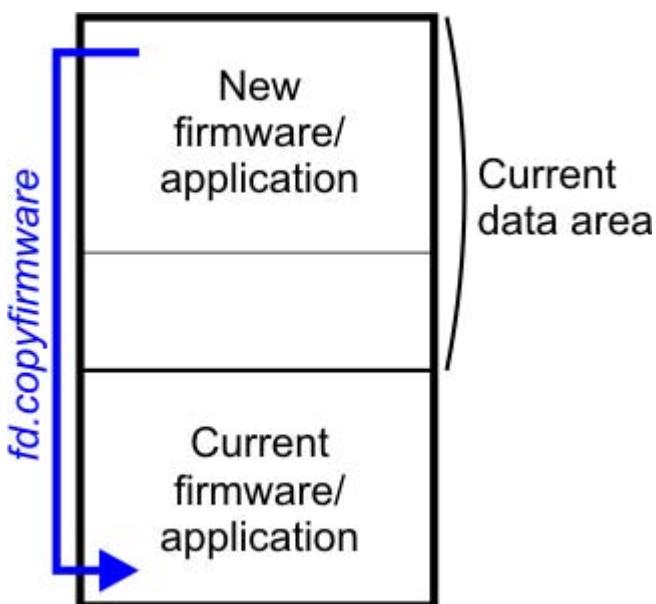
Notice how the i variable in the example above is not doing anything useful. The csum argument of the fd.checksum method exists to return the result of the checksum verification or calculation. For example, it can allow you not only to set correct checksum for the RAM buffer, but also find out what the checksum is. Who knows... you might need it.

Upgrading the Firmware/Application

As was explained under [Direct Sector Access](#)^[448], you can't use [fd.setsector](#)^[483] to write to the flash area occupied by the firmware and Tibbo Basic application. This is done to prevent your application from inadvertently damaging its own code or the firmware.

A special [fd.copyfirmware](#)^[460] method is provided for changing the data in the firmware/application area of the flash. The method copies the specified number of sectors (starting from the sector with logical number 0) from the data area and into the firmware/application area of the flash memory, then reboots your device.

Before invoking this method, store the new binary image of the firmware/application in the data area of the flash memory starting from logical sector #0. Logical numbers are assigned to the sectors in reverse, so storing to sector #0 actually means storing to the topmost physical sector of the flash memory. It goes without saying that your data area *has* to have enough capacity to store the new binary image.



Exactly how you receive the new binary image is immaterial to this discussion. You can use any suitable transmission method, such as TCIP/IP, FTP, your own proprietary protocol, whatever! The important part is that before invoking `fd.copyfirmware` you must have the binary image stored in the data area of the flash, starting from logical sector #0, and you must know how many sectors this occupies. After that, invoke `fd.copyfirmware` and hope that you prepared the right data -- **wrong data will render your device inoperable**. That's right -- no second chance for a remote upgrade if you loaded a wrong binary! You will have to physically go to your device and upload it with the right firmware and it is quite possible that you will have to do this through the serial port.

File-based and Direct Sector Access Coexistence

[File-based](#)^[436] and [direct sector](#)^[448] access methods can be used at the same time, as long as you understand how direct access may affect (and possibly *screw up*) the flash disk.

[Direct Sector Access](#)^[448] explains that working with flash sectors is done through two RAM buffers numbered #0 and #1. Flash disk operation depends on these buffers as well.

Buffer #0 is used for processing housekeeping data and stores no valuable content that must be preserved past the end of a certain disk-related method execution. That is, once the method such as the [fd.setdata](#)^[480] has executed, buffer #0 has no valuable data in it.

Buffer #1 is used to load and store the contents of the most recently loaded (and possibly changed) data sector. So, if fd.setdata was recently called, this buffer may still hold new data, and this data may not be saved to flash yet. Corrupting the data would have unpleasant consequences for the file. To prevent this, the fd. object automatically dismounts the disk (sets the [fd.ready](#)^[476]= 0- NO) if your application does [fd.setbuffer](#)^[479] or [fd.getsector](#)^[472] while the [fd.buffernum](#)^[458]= 1 and while this buffer was loaded with *new and as yet unsaved* sector data.

Preventing disk dismounting is easy. You can opt to work on the RAM buffer #0 (set fd.buffernum= 0), or flush the unsaved data by invoking the [fd.flush](#)^[466] or [fd.close](#)^[460] method.

The fd.setsector method will cause disk dismounting if the destination was within the disk area (0 - fd.totalsize-1). Writing outside the disk area will not interfere with the flash disk operation.

The [fd.getbuffer](#)^[468] method does not cause any interference with the flash disk, so it can be used freely.

The [fd.checksum](#)^[459] will write to the RAM buffer (when in the "PL_FD_CSUM_MODE_CALCULATE" mode), but the checksum calculation method is the same as the one used by the disk itself, so setting this checksum will never be wrong.

Note, finally, that any file-related method will affect the value of the fd.buffernum. Never assume that you know what this property is set to. Always set it explicitly.

Prolonging Flash Memory Life

Flash memory has a huge limitation -- the number of times you can rewrite each of its sectors is limited to around 100'000 times. At first this may seem like a virtually unreachable limit, but in reality you can "get there" quite fast, which is not a good thing. Once the flash wears down, you will start having data errors, data corruption, failed sectors, etc. This topic explains what you can do to prolong the life of the flash memory used in your device. This is generally achieved by (1) reducing the number of writes to the flash memory, and (2) using sector leveling, i.e. spreading sector writes, as evenly as possible, between the sectors.

File-based access

For [file-based access](#)^[436], the fd. object does the bulk of work for you. We have already [explained](#)^[437] that the file record table (FRT) and the file allocation table (FAT) -- the most heavily written to areas of the disk -- occupy at least double the space needed to store their data. They also have the minimum number of sectors they will take, regardless of what is necessary. This ensures that at any given time, the FRT and the FAT have spare sectors. These spare sectors, as well as occupied sectors, are in constant rotation. For example, every time you create a file, a change is made to one of the sectors of the FRT. This change requires

writing data to one of the physical sectors. In the process, the fd. object will take the previously used FRT sector and "release it" into the pool of spare FRT sectors. At the same time, one of the spare FRT sectors will become active and store changed data. The FAT operates in the same manner. While being fully transparent to your application, the process greatly prolongs useful flash memory life.

You can, and are advised to, further reduce the wear of the FAT and FRT by decreasing the number of writes that will be required. One way to do so is to create all necessary files and allocate space for them once -- typically when your application "initializes" your device.

Say, you have a log file, which stores events registered by your application. An obvious approach would be to simply append each new event's data to the file. This way, the file will grow with each event added. But wait a second, this means that the FRT area, which keeps current file size, will be changed each time you add to the file! The FAT area will be stressed too!

An alternative approach would have us create a file of desired maximum size once and fill it up with "blank" data (such as &hFF codes). We will then overwrite this blank data with actual event data as events are generated. This time around, our actions will be causing no changes in the FRT and FAT areas, thus prolonging the life of the flash IC. Incidentally, this approach is also more [reliable](#)⁴⁵⁴.

The second method is, of course, more complicated. For example, you will need to remember or be able to detect where in the file the new event will go, rather than simply append the event to the end of the file. The benefits, however are plentiful and the effort is worthwhile.

The data area of the disk has limited leveling that results in spreading unused sector utilization. The fd. object makes sure that when your file needs a new data sector, this data sector will be selected from a pool of available data sectors in a random fashion. Once the data sector has been allocated to a file, however, it stays with that file for as long as necessary. So, if you are writing at a certain file offset over and over again, you are stressing the same physical sector of the flash IC.

On large files, you rarely write at the same offset all the time. For example, if you have a log file that has 1000 data sectors, then it is unlikely you will be writing to the same sector over and over again. For smaller files, the probability is higher. Your solution is to, from time to time (not too often), erase the file and recreate it again. This will randomly allocate new sectors for the file.

Direct sector access

[Direct sector access](#)⁴⁴⁸ is a low-level form of working with the flash. You are your own master, the fd. object does not help you with anything, and it is up to you to make sure that the flash IC is not being worn out unevenly. Generally speaking, limit the number of times you are writing to the flash and/or implement some form of leveling where a large number of sectors are used to share the same task and each sector gets its fair share of work.

Ensuring Disk Data Integrity

Maintaining the data integrity is a very important task, and the one where the flash memory needs a lot of help from your smart Tibbo Basic application. The biggest source of potential trouble is a sudden loss of power right in the middle of writing to the flash IC. This can cause devastation on two levels:

- The data in flash sectors is changed by first erasing the sector (a process in which all sector locations return to the value of &hFF), and then writing the new data. Should the power fail right in the middle of this process, you may end up

losing the old and the new contents of the sector.

- For many [flash disk operations](#)^[436], a single operation requires making changes to several sectors. For example, creating a file requires to change data both in the file record table (FRT) and the file allocation table (FAT). Power failure in the middle of this process can leave the disk with the FRT already changed, and the FAT not yet changed -- a "loss of sync" situation that renders the disk unusable.

Here is how you can address potential integrity issues:

File-based access

First and foremost -- the less you change the data, the more secure your data is. Your flash memory life is [prolonged](#)^[453] this way, too.

Secondly, try limiting rewrites in the FRT and FAT -- these most sensitive areas of the disk. [Prolonging Flash Memory Life](#)^[453] explains how creating all the files and setting their size once helps keep your flash memory healthy. The same approach also makes your system more reliable -- if the FAT/FRT does not get changed, then the fatal data corruption in FRT and/or FAT will not happen. If, however, you do need to create or delete the files, you can backup the entire service area of the disk first. Should the power fault occur in the middle of making changes to the FAT/FRT, you can simply restore them to their previous state.

To use backups, make your flash disk occupy a bit less space than available -- just enough for the backup of the entire "housekeeping" area of your flash disk to fit. The calculation for the housekeeping area size can be found in the [Disk Area Allocation Details](#)^[437] topic.

Before creating/deleting files, make a backup copy of the FAT/FRT data:

```
Dim f As Byte
...
'backup disk housekeeping data
fd.buffernum=0
For f=0 To fd.numservicesectors-1
    fd.getsector(f)
    fd.setsector(fd.totalsize+f)
Next f
```

Should the need arise, you can copy the data back. Of course, you will need to store the number of sectors to copy back somewhere, for example, in the EEPROM ([.stor](#)^[380] object). You will also probably have a "transaction flag". If your device boots up and the flag is set, then the power failure has occurred in the middle of an important disk "transaction" and the previous state must be restored.

The power disaster may also strike when you are changing the data in the file itself. The [fd.sector](#)^[478] R/O property always tells you the number of a sector corresponding to the current pointer position ([fd.pointer](#)^[476]). This way you can backup this sector first and, should the power failure occur right in the middle of the data alteration, restore the original sector contents.

Note, of course, that writing to a file can span two sectors. Depending on the pointer position, your write could begin in one sector and end in the next. The next sector is also not simply the `fd.sector+1`, because the sectors belonging to the same file may actually be scattered across the disk. So, to make sure your sector backup works, make sure the sector boundaries are never crossed during writing. This can be easily achieved if you are dealing with "data table records" and record sizes are limited to the power of 2 (i.e. 1, 2, 4, 8, 16, 32, 64, 128 bytes per record).

Direct sector access

Again, you are your own master. Do whatever you want with the flash memory. Just remember to always plan for potential power failure. It can happen at any time, and you better figure out how your application will recover the data.

Properties and Methods

The following classification groups properties and methods of the fd. object by their logical function.

Properties and methods under [fd.filenum](#)^[463] are indented to reflect the fact that they are performed on the file currently selected by the fd.filenum.

"**D**" means that changing the value of the property or executing the method will dismount the disk.

"**M**" means that the disk must be mounted ([fd.mount](#)^[474]) before this method can be successfully executed or the property will return a meaningful data.

"**S**" means that executing the method affects [fd.laststatus](#)^[472] and *may* also affect [fd.ready](#)^[476]. In other words, method execution may fail, so the fd.laststatus will carry the error code, and, depending on the error, the disk may be dismounted.

"**O**" means that the file must be opened (on a currently selected [fd.filenum](#)^[463]) for the method to be successfully executed or the property to return a meaningful data.

Direct sector access:

- [fd.buffernum](#)^[458] [Property, **D**]
- [fd.getsector](#)^[472] [Method, **D**]
- [fd.setsector](#)^[483] [Method, **D**]
- [fd.getbuffer](#)^[468] [Method, **D**]
- [fd.setbuffer](#)^[479] [Method, **D**]
- [fd.checksum](#)^[459] [Method]
- [fd.copyfirmware](#)^[460] [Method]

General disk info and operations:

- [fd.availableflashspace](#)^[457] [R/O Property]
- [fd.maxopenedfiles](#)^[473] [R/O Property]
- [fd.format](#)^[467] [Method, **D, S**]
- [fd.mount](#)^[474] [Method, **S**]
- [fd.numservicesectors](#)^[475] [R/O Property, **M**]
- [fd.capacity](#)^[458] [R/O Property, **M**]
- [fd.totalsize](#)^[483] [R/O Property, **M**]
- [fd.getfreespace](#)^[470] [Method, **M-S**]

- [fd.laststatus](#)^[472] [R/O Property]
- [fd.ready](#)^[476] [R/O Property]

Directory-related:

- [fd.maxstoredfiles](#)^[474] [R/O Property, M]
- [fd.getnumfiles](#)^[471] [Method, M-S]
- [fd.resetdirpointer](#)^[478] [Method]
- [fd.getnextdirmember](#)^[470] [Method, M-S]
- [fd.getattributes](#)^[467] [Method, M-S]
- [fd.setattributes](#)^[478] [Method, M-S]
- [fd.create](#)^[462] [Method, M-S]
- [fd.delete](#)^[463] [Method, M-S]
- [fd.open](#)^[475] [Method, M-S]
- [fd.flush](#)^[466] [Method, M-S]

File access:

- [fd.filenum](#)^[463] [Property]
 - [fd.open](#)^[475] [Method, M-S]
 - [fd.close](#)^[460] [Method, M-S-O]
 - [fd.fileopened](#)^[464] [R/O Property]
 - [fd.filesize](#)^[464] [R/O Property, M-O]
 - [fd.cutfromtop](#)^[461] [Method, M-S-O]
 - [fd.setfilesize](#)^[481] [Method, M-S-O]
 - [fd.pointer](#)^[476] [R/O Property, M-O]
 - [fd.setpointer](#)^[482] [Method, M, S-O]
 - [fd.getdata](#)^[469] [Method, M-S-O]
 - [fd.setdata](#)^[480] [Method, M-S-O]
 - [fd.find](#)^[464] [Method, M-S-O]
 - [fd.sector](#)^[478] [R/O Property, M-O]

.Availableflashspace R/O Property

Function:	Returns the total number of sectors available to store application's data.
Type:	Word
Value Range:	Value depends on the flash capacity and firmware/application size.
See Also:	Sharing Flash Between Your Application and Data ^[434] , fd.totalsize ^[483]

Details

The value of this property reflects free flash space not occupied by currently loaded Tibbo Basic application. You may use the [fd.format](#)^[467] method to create the flash disk as large as available flash space. Note that the actual capacity of the disk ([fd.capacity](#)^[458]) will be less than the available space because the flash disk also needs a certain number of sectors for its "housekeeping" data (see [fd.numservicesectors](#)^[475]).

.Buffernum Property

Function:	Sets/returns the number of the currently selected RAM buffer.
Type:	Byte
Value Range:	0 or 1. Default = 0 (RAM buffer #0 selected).
See Also:	---

Details

There are two 264-byte RAM buffers used to exchange the data with physical sectors of the flash memory -- #0 and #1. This property selects the buffer that will be used as the data destination for [fd.setbuffer](#)^[479] and [fd.getsector](#)^[472] methods, or the data source for [fd.getbuffer](#)^[468], [fd.setsector](#)^[483], and [fd.checksum](#)^[459] methods.

All file-based operations of the flash disk rely on these two RAM buffers too and selected buffer number may change as a result of their execution. When using [direct sector access](#)^[448] and [file-based access](#)^[436] concurrently, switch to the RAM buffer #0 for direct sector access -- this will guarantee that you won't corrupt the data of currently opened files.

.Capacity R/O Property

Function:	Returns the capacity of the currently existing flash disk in sectors.
Type:	Word
Value Range:	0-65535
See Also:	Checking Disk Vitals ^[444] , fd.format ^[467]

Details

Capacity reflects the number of usable sectors on the disk. Total number of sectors occupied by the disk (see [fd.totalsize](#)^[483]) in flash memory is larger and also includes "housekeeping" sectors (see [fd.numservicesectors](#)^[475]). Currently existing disk may occupy less space than it potentially could (see [fd.availableflashspace](#)^[457]).

The disk must be mounted (see [fd.mount](#)^[474]) for this property to return a meaningful value.

Checksum Method

Function:	Calculates or verifies the checksum for the data in the currently selected RAM buffer of the flash memory (selection is made through the fd.buffernum ^[458] property).
Syntax:	fd.checksum(mode as pl_fd_csum_mode, byref csum as word) as ok_ng
Returns:	0- OK: Completed successfully (always the case when the mode= 1- PL_FD_CSUM_MODE_CALCULATE). 1- NG : The checksum was found to be invalid (can only be generated when the mode= 0- PL_FD_CSUM_MODE_VERIFY). Also returns the calculation result indirectly, through the csum argument.
See Also:	Using Checksums ^[450] , fd.getsector ^[472] , fd.setsector ^[483] , fd.getbuffer ^[468] , fd.setbuffer ^[479]

Par	Description
mo	0- PL_FD_CSUM_MODE_VERIFY: verify the checksum.
de	1- PL_FD_CSUM_MODE_CALCULATE: calculate the checksum.
csu	Indirectly returns calculated value. When the mode= 0-
m	PL_FD_CSUM_MODE_VERIFY, this value returned will be 0 if the checksum was found to be correct, or some other value if the checksum was found to be wrong. When the mode= 1- PL_FD_CSUM_MODE_CALCULATE, the method will return a newly calculated checksum.

Details

Each physical sector of the flash memory has 264 bytes of data. The flash disk of the .fd object uses 256 bytes to store the data, and two of the remaining 8 bytes (at offsets 256 and 257, counting from 0) are used to store the checksum of the data. This is an arbitrary choice -- there is no special reason why the .fd object does it this way, and not in any other way.

During checksum verification, first 258 bytes of the selected RAM buffer (data + checksum) are added together as 129 sixteen-bit values. The result is then limited to lower 16 bits. If the value is 0, then the checksum is correct. If the value is not 0, then the checksum is incorrect.

During checksum calculation, first 256 bytes of the selected RAM buffer are added together as 128 sixteen-bit values. The result is then limited to lower 16 bits and subtracted from &h10000. Obtained value is the checksum which is subsequently stored at offsets 256 and 257 of the RAM buffer.

16-bit values for these calculations are little-endian. That is, offset 0 of the RAM buffer is presumed to be the high byte of the first 16-bit value, offset 1 -- low byte

of the first 16-bit value, offset 2 -- high byte of the second 16-bit value, and so on.

.Close Method

Function:	Closes the file opened "on" a currently selected file number (selection is made through the fd.filenum ^[463]).
Syntax:	fd.close() as pl_fd_status_codes
Returns:	One of the following pl_fd_status_codes ^[472] , also affects fd.laststatus ^[472] :
	0- PL_FD_STATUS_OK: Completed successfully. 1- PL_FD_STATUS_FAIL: Physical flash memory failure (fatal).

See Also: [Closing Files](#)^[448]

Details

Invoking the method also does the job performed by the [fd.flush](#)^[466] method (and this is why the 1- PL_FD_STATUS_FAIL status code may be returned here). Attempting to invoke this method "on" a file number that did not have any opened file associated with it generates no error.

.Copyfirmware Method

Function:	Copies specified number of sectors, starting from logical sector 0, into the firmware and Tibbo Basic area of the flash memory, then reboots the device to make it run the new firmware.
Syntax:	fd.copyfirmware(numsectors as word)
Returns:	---
See Also:	Upgrading the Firmware/Application ^[452] , fd.buffernum ^[458] , fd.getbuffer ^[468] , fd.getsector ^[472]

Part	Description
num	Number of sectors to copy.
sect	
ors	

Details

This method allows you to remotely upgrade the firmware and the Tibbo Basic application of your device. First, your application can receive and store the new firmware into the unused area of the flash memory (see [fd.availableflashspace](#)^[457], [fd.setbuffer](#)^[479], [fd.setsector](#)^[483]). Actual transmission of the firmware/application can be implemented in any suitable way, for example, through a TCP/IP connection. To upgrade itself, your application will then execute this method.

BE VERY CAREFUL! Using the `fd.copyfirmware` on incorrect data will "incapacitate" your device and further remote upgrades will become impossible.

You will need to physically go to your device and upload correct firmware through its serial port.

.Cutfromtop Method

Function:	Removes a specified number of sectors from the beginning of a file opened "on" a currently selected file number (selection is made through the fd.filenum ^[463]).
Syntax:	fd.cutfromtop(numsectors as dword) as pl_fd_status_codes
Returns:	One of the following pl_fd_status_codes ^[472] , also affects fd.laststatus ^[472] :
	0- PL_FD_STATUS_OK: Completed successfully. 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal). 2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal). 3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal). 8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

See Also: [Removing Data From Files](#)^[445]

Part	Description
------	-------------

num sector rs	Number of sectors to remove from the beginning of the file. Supplied value will be corrected if exceeded total number of sectors allocated to this file.
---------------------	--

Details

While the [fd.setfilesize](#)^[481] allows to remove data from the end of the file, the [fd.cutfromtop](#) allows to remove data from the beginning of the file. Removed data sectors previously allocated to this file will be "released" (marked unused) as they become unnecessary. The file, however, will always have at least one data sector allocated to it. That is, if the file occupied three sectors, and you do [fd.cutfromtop\(3\)](#), then one data sector will still remain in this file.

The size of the file (see [fd.filesize](#)^[464]) will be corrected downwards in accordance with the amount of removed data. For example, performing [fd.cutfromtop\(2\)](#) on a file occupying 3 data sectors will reduce its size by 512 bytes (amount of data in 2 sectors removed). Performing [fd.cutfromtop\(3\)](#) will set the file size to 0.

The pointer position (see [fd.pointer](#)^[476]) is always reset as a result of this method execution. If the new file size is 0, then the pointer will be set to 0 as well. If the file is not empty, then the pointer will be set to 1.

All possible status codes returned by this method are "generic" and are not described here.

Create Method

Function:	Creates a new file with the specified name and attributes.
Syntax:	fd.create(byref name_attr as string) as pl_fd_status_codes
Returns:	<p>One of the following pl_fd_status_codes^[472], also affects fd.laststatus^[472]:</p> <ul style="list-style-type: none"> 0- PL_FD_STATUS_OK: Completed successfully. 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal). 2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal). 4- PL_FD_STATUS_INV_PARAM: Invalid argument have been provided. 5- PL_FD_STATUS_DUPLICATE_NAME: File with this name already exists. 6- PL_FD_STATUS_FILE_TABLE_FULL: Maximum number of files that can be stored on the disk has been reached, new file cannot be created. 7- PL_FD_STATUS_DATA_FULL: The disk is full, new data cannot be added. 8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

See Also: [Creating and Deleting Files](#)^[441]

Par t	Description
na	A string (1-56 characters), must contain a file name and, optionally,
me	attributes separated from the file name by a space. File names are case-
_at	sensitive.
tr	

Details

The file_attr string will always be truncated to 56 characters. Leading spaces will be removed. The file name must contain at least one character, or 4-PL_FD_STATUS_INV_PARAM code will be returned. The file name must be unique, attempting to create another file with the same name will lead to the 5-PL_FD_STATUS_DUPLICATE_NAME status code. Note that only the name must be unique, the attributes can be the same for several files on the disk. Any character except space can be used in file names. This includes "/" and "\". The flash disk does not support subdirectories, but it is possible to emulate them by including "/" or "\"" characters in the file name. The attributes portion of the string may contain any characters whatsoever.

The flash disk can accommodate a limited number of files, which is determined during the formatting (see [fd.format](#)^[467]) and can be checked through the [fd.maxstoredfiles](#)^[474] R/O property. Attempting to create another file when the maximum number of stored files has already been reached will result in the 6-PL_FD_STATUS_FILE_TABLE_FULL status code.

Each file always occupies at least one sector in the data area of the disk. The first sector is allocated when you invoke the fd.create method. This is why an attempt to create a new file may result in the 7- PL_FD_STATUS_DATA_FULL status code. All other possible error states are "generic" and are not described here.

.Delete Method

Function:	Deletes a file from the flash disk.
Syntax:	fd.delete(byref name as string) as pl_fd_status_codes
Returns:	One of the following pl_fd_status_codes ^[472] , also affects fd.laststatus ^[472] :
	0- PL_FD_STATUS_OK: Completed successfully.
	1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).
	2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).
	3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).
	8- PL_FD_STATUS_NOT_READY: The disk is not mounted.
	9- PL_FD_STATUS_NOT_FOUND: File not found.
See Also:	Creating and Deleting Files ^[441]

Par t	Description
na	A string (1-56 characters) with the file name. All characters after the first space encountered (excluding leading spaces) will be ignored. File names are case-sensitive.
me	

Details

The file string will always be truncated to 56 characters. Leading spaces will be removed. The file with the specified name must exist or the 9- PL_FD_STATUS_NOT_FOUND status code will be returned. It is OK to delete a file which is currently opened -- after that, the [fd.fileopened](#)^[464] for this file number will be reset to 0- NO.

All other possible error states are "generic" and are not described here.

.Filenum Property

Function:	Sets/returns the number of the currently selected file.
Type:	Byte
Value Range:	0 to fd.maxopenedfiles ^[473] -1. Default= 0 (file #0 selected).
See Also:	Opening Files ^[443]

Details

Several files can be opened (see [fd.open](#)^[475]) at the same time. Each file is said to be opened "on" a certain file number (the value of this property at the time of the file opening). Although the file is opened by referring to its name, many other operations, such as [fd.setdata](#)^[480] or [fd.close](#)^[460], as well as properties, such as [fd.filesize](#)^[464], are related to the file number selected through the fd.filenum.

.Fileopened R/O Property

Function:	Reports if any file is currently opened "on" the selected file number (selection is made through the fd.filenum ^[463]).
Type:	Enum (no_yes, byte)
Value Range:	0- NO: No file is currently opened on this file number (default). 1- YES: The file is currently opened on this file number.
See Also:	Opening Files ^[443]

Details

Use the [fd.open](#)^[475] method to open a file "on" the currently selected file number. Many file-related operations, such as [fd.setdata](#)^[480] or [fd.close](#)^[460], as well as properties, such as [fd.filesize](#)^[464], cannot be used unless there is an opened file on a currently selected file number.

.Filesize R/O Property

Function:	Returns the size, in bytes, of the file opened "on" the currently selected file number (selection is made through the fd.filenum ^[463] or zero if no file is currently opened).
Type:	Dword
Value Range:	0- whatever is physically possible for the currently existing flash disk.
See Also:	Writing To and Reading From Files ^[444]

Details

Newly created files have the size of 0. File size can be altered by adding data to the file ([fd.setdata](#)^[480]), or reducing the file size with [fd.setfilesize](#)^[481] and [fd.cutfromtop](#)^[461] methods.

.Find Method

Function:	Finds Nth instance of data satisfying selected criteria in a file opened "on" a currently selected file number (selection is made through the fd.filenum ^[463]).
------------------	--

Syntax:

```
fd.find(frompos as dword, byref substr as string,
instance as word, dir as forward_back, incr as word,
mode as pl_fd_find_modes) as dword
```

Returns:

File position (counting from one) at which the target occurrence of the substr was discovered, or 0 if the target occurrence of the substr was not found.

The method also affects the state of the [fd.laststatus](#)⁴⁷². The following status codes are possible:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).

3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).

4- PL_FD_STATUS_INV_PARAM: Invalid argument have been provided.

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

See Also:

[Searching Files](#)⁴⁴⁶

Par t	Description
fro	Starting position in a file from which the search will be conducted. File positions are counted from 1. Will be corrected automatically if out of range.
mp	
os	
sub	The search criteria.
str	
inst	Instance (occurrence) number to find.
anc	
e	
dir	Search direction: 0- FORWARD: the search will be conducted from the frompos position and towards the end of the file. 1- BACK: the search will be conducted from the frompos position and towards the beginning of the file.
incr	Search position increment (or decrement for BACK searches).

mo Search mode:
 de 0- PL_FD_FIND_EQUAL: Find data that is equal to the substr.
 1- PL_FD_FIND_NOT_EQUAL: Find data that is not equal to the substr.
 2-PL_FD_FIND_GREATER: Find data with value greater than the value of the substr.
 3- FIND_GREATER_EQUAL: Find data with value greater than or equal to the value of the substr.
 4- PL_FD_FIND_LESSER: Find data with value less than the value of the substr.
 4- PL_FD_FIND_LESSER_EQUAL: Find data with value less than or equal to the value of the substr.

Details

The fd.find is a powerful method that allows you to do the "full text" search (incr=1) or "database record" search (incr= your_record_size) across the file. Searching back is less efficient than searching forward (by ~50%).

All status codes generated by this method are "generic" and are not described here.

.Flush Method

Function:	Saves back to the flash memory ("flushes") the changes made to the most recently edited file.
Syntax:	fd.close() as pl_fd_status_codes
Returns:	One of the following pl_fd_status_codes ^[472] , also affects fd.laststatus ^[472] : 0- PL_FD_STATUS_OK: Completed successfully. 1- PL_FD_STATUS_FAIL: Physical flash memory failure (fatal).
See Also:	Writing To and Reading From Files ^[444]

Details

When any sector of the flash disk is being altered, this is done in a RAM buffer. To improve efficiency and reduce sector wear, the data from the RAM buffer is written back to the flash memory only when it becomes necessary to load the buffer with the contents of another sector. When the file is closed (see [fd.close](#)^[460]), buffer "flushing" is done automatically. However, if changes are made to any file and then no disk activity is performed for this or any other file afterwards, the buffer may keep the last changes made indefinitely. These changes will be lost if your device reboots. To prevent this, use the fd.flush method.

Note that the fd.flush method does not depend on the current [fd.filenum](#)^[463] value and works globally on any most recently changed file.

.Format Method

Function:	Formats the flash memory to create a flash disk.
Syntax:	fd.format(totalsize as word, numstoredfiles as byte) as pl_fd_status_codes
Returns:	One of the following pl_fd_status_codes ^[472] , also affects fd.laststatus ^[472] : 0- PL_FD_STATUS_OK: Completed successfully. 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal). 4- PL_FD_STATUS_INV_PARAM: Invalid argument have been provided for the invoked method.
See Also:	Formatting the Flash Disk ^[436]

Part	Description
totalsize	Desired number of sectors occupied by the disk in flash memory. Cannot exceed available space (fd.availableflashspace ^[457]) and may be slightly corrected downwards for internal housekeeping reasons. Actual total size can be checked through the fd.totalsize ^[483] .
maxstoredfiles	Desired maximum number of files that the disk will allow to create. Actual maximum number of files will be adjusted automatically to be a multiple of four and not exceed 64 (for example, specifying 6 will result in the actual value of 8). If you specify 0 you will get 4 files.

Details

Actual usable capacity ([fd.capacity](#)^[458]) of the formatted disk will be less than its total size. This is because the disk also includes "housekeeping" sectors (see [fd.numservicesectors](#)^[475]). Reducing the maxstoredfiles parameter decreases the number of required housekeeping sectors.

The 4- PL_FD_INV_PARAM status code will be returned if the totalsize specified exceeded available space ([fd.availableflashspace](#)) or was too small even for the "housekeeping" data of the disk to fit in.

After formatting the disk will be in the dismounted state and will need to be mounted (see [fd.mount](#)^[474]) before any disk-related activity can be successfully performed.

.Getattributes Method

Function:	Returns the attributes string for a file with the specified file name.
Syntax:	fd.getattributes(byref name as string) as string

Returns:

The string with attributes (may be up to 54 characters long) or an empty string if no attributes were set for this file. The method also affects the state of the [fd.laststatus](#)^[472]. The following status codes are possible:

- 0- PL_FD_STATUS_OK: Completed successfully.
- 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).
- 2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).
- 3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).
- 8- PL_FD_STATUS_NOT_READY: The disk is not mounted.
- 9- PL_FD_STATUS_NOT_FOUND: File not found.

See Also:

[Reading and Writing File Attributes](#)^[442]

Part	Description
------	-------------

name	A string (1-56 characters) with the file name. All characters after the first space encountered (excluding leading spaces) will be ignored. File names are case-sensitive.
------	--

Details

The length of returned data will automatically be truncated to the capacity of the receiving string variable.

The attributes string can be defined for the file when creating the file using the [fd.create](#)^[462] method or set afterwards with the [fd.setattributes](#)^[478] method.

This method returns its execution status indirectly, through the fd.laststatus R/O property. The file with the specified name must exist or the 9-PL_FD_STATUS_NOT_FOUND status code will be returned. All other possible error states are "generic" and are not described here.

.Getbuffer Method**Function:**

Reads a specified number of bytes from the currently selected RAM buffer of the flash memory (selection is made through the [fd.buffernum](#)^[458] property).

Syntax:

fd.getbuffer(offset as word, len as word) as string

Returns:

The string with the data read from the buffer. The method also affects the state of the [fd.laststatus](#)^[472]. The following status codes are possible:

- 0- PL_FD_STATUS_OK: Completed successfully.
- 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

See Also:

[Direct Sector Access](#)^[448], [fd.setbuffer](#)^[479], [fd.getsector](#)^[472], [fd.setsector](#)^[483], [fd.checksum](#)^[459]

Par t	Description
----------	-------------

offs Starting offset in the buffer. Possible value range is 0-263 (the buffer stores 264 bytes of data, offset is counted from 0).
 et
 len Number of bytes to read. Will be corrected downwards if necessary.

Details

The length of returned data will depend on one of three factors, whichever is smaller: len argument, amount of data still available in the buffer counting from the *offset* position, and the capacity of receiving string variable.

All status codes generated by this method are "generic" and are not described here.

.Getdata Method

Function:

Reads a specified number of bytes from the file opened "on" a currently selected file number (selection is made through the [fd.filenum](#)^[463]).

Syntax:

fd.getdata(maxinplen as byte) as string

Returns:

The string with the data read from the file. The method also affects the state of the [fd.laststatus](#)^[472]. The following status codes are possible:

- 0- PL_FD_STATUS_OK: Completed successfully.
- 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).
- 2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).
- 3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).
- 8- PL_FD_STATUS_NOT_READY: The disk is not mounted.
- 10- PL_FD_STATUS_NOT_OPENED: No file is currently opened "on" the current value of the [fd.filenum](#)^[463] property.

See Also:

[Writing To and Reading From Files](#)^[444], [fd.setpointer](#)^[482], [fd.setdata](#)^[480]

Par t	Description
----------	-------------

ma Maximum number of bytes to read from the file.
 xin
 ple
 n

Details

The length of returned data will depend on one of three factors, whichever is smaller: maxinplen argument, amount of data still available in the file counting from the current pointer position (see [fd.pointer](#)^[476]), and the capacity of receiving string variable.

As a result of this method invocation, the pointer will be advanced forward by the number of bytes actually read from the file.

All status codes generated by this method are "generic" and are not described here.

.Getfreespace Method

Function:	Returns the total number of free data sectors available on the flash disk.
Syntax:	fd.getfreespace() as word
Returns:	0-65535, also affects the state of the fd.laststatus ^[472] . The following status codes are possible: 0- PL_FD_STATUS_OK: Completed successfully. 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal). 2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal). 3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal). 8- PL_FD_STATUS_NOT_READY: The disk is not mounted.
See Also:	Checking Disk Vitals ^[441] , fd.capacity ^[458] , fd.numservicesectors ^[475] , fd.totalsize ^[483]

Details

This method returns its execution status indirectly, through the fd.laststatus R/O property. All possible error states are "generic" and are not described here.

.Getnextdirmember Method

Function:	Returns the next filename (if any) found in the disk directory.
Syntax:	fd.getnextdirmember() as string

Returns:

The string containing the file name of the next directory member or an empty string if all file names have already been returned. The method also affects the state of the [fd.laststatus](#)^[472]. The following status codes are possible:

- 0- PL_FD_STATUS_OK: Completed successfully.
- 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).
- 2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).
- 3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).
- 8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

See Also:

[Walking Through File Directory](#)^[442]

Details

The length of returned data will automatically be truncated to the capacity of the receiving string variable.

Each time you invoke this method, internal directory "pointer" is incremented by one. Therefore, repeated invocation of the method will allow you to read out the names of all files currently found on the disk. Use [fd.resetdirpointer](#)^[473] beforehand to move the pointer back to zero. To get the names of all files, you can invoke the fd.getnextdirmember repeatedly until the empty string is returned or for the fd.getnumfiles number of times.

This method returns its execution status indirectly, through the fd.laststatus R/O property. All possible status codes generated by this method are "generic" and are not described here.

.Getnumfiles Method

Function:

Returns the total number of files currently stored on the disk.

Syntax:

fd.getfreespace() as word

Returns:

0-65535, also affects the state of the [fd.laststatus](#)^[472]. The following status codes are possible:

- 0- PL_FD_STATUS_OK: Completed successfully.
- 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).
- 2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).
- 3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).
- 8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

See Also:

[Walking Through File Directory](#)^[442], [fd.maxstoredfiles](#)^[474]

Details

This method returns its execution status indirectly, through the fd.laststatus R/O property. All possible status codes generated by this method are "generic" and are not described here.

.Getsector Method

Function:	Reads a specified sector into the currently selected RAM buffer of the flash memory (selection is made through the fd.buffernum ^[458] property).
Syntax:	fd.getsector(num as word) as pl_fd_status_codes
Returns:	One of the following pl_fd_status_codes ^[472] , also affects fd.laststatus ^[472] :
	0- PL_FD_STATUS_OK: Completed successfully. 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

See Also: [Direct Sector Access](#)^[448], [fd.setsector](#)^[483], [fd.getbuffer](#)^[468], [fd.setbuffer](#)^[479], [fd.checksum](#)^[459]

Par	Description
t	

nu Sector to read from.
m

Details

Each physical sector of the flash memory has 264 bytes of data, and all 264 bytes will be loaded into the RAM buffer.

Note that reading the sector into the RAM buffer #1 may interfere with file-based access to the flash memory. If you have executed [fd.setdata](#)^[480], the RAM buffer #1 may still contain this new data, in which case using the fd.getsector while the fd.buffernum= 1 will automatically dismount the disk (set [fd.ready](#)^[476]= 0- NO). To avoid this situation, switch to the RAM buffer #0, or use [fd.flush](#)^[466] or [fd.close](#)^[460] methods before invoking the fd.getsector.

All status codes generated by this method are "generic" and are not described here.

.Laststatus R/O Property

Function:	Returns the execution result for the most recent disk-related method execution.
Type:	Enum (pl_fd_status_code, byte)

Value Range:	0- PL_FD_STATUS_OK: Completed successfully (default)
	1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).
	2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).
	3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).
	4- PL_FD_STATUS_INV_PARAM: Invalid argument have been provided for the invoked method.
	5- PL_FD_STATUS_DUPLICATE_NAME: File with this name already exists.
	6- PL_FD_STATUS_FILE_TABLE_FULL: Maximum number of files that can be stored on the disk has been reached, new file cannot be created.
	7- PL_FD_STATUS_DATA_FULL: The disk is full, new data cannot be added.
	8- PL_FD_STATUS_NOT_READY: The disk is not mounted.
	9- PL_FD_STATUS_NOT_FOUND: File not found.
	10- PL_FD_STATUS_NOT_OPENED: No file is currently opened "on" the current value of the fd.filenum ^[463] property.
	11- PL_FD_STATUS_ALREADY_OPENED: This file is already opened "on" some other file number.

See Also:[Fd. Object's status codes](#)^[435]**Details**

Some methods, such as [fd.create](#)^[462], return execution status directly. For those, the fd.laststatus will contain the same status as the one directly returned.

Some methods, such as [fd.getdata](#)^[469] return some other data, or nothing in case there was a problem. The execution result for such methods can only be verified through the fd.laststatus property.

Note that some errors are fatal and the disk is dismounted ([fd.ready](#)^[476] is set to 0-NO) immediately upon the detection of any such fatal error.

.Maxopenedfiles R/O Property

Function: Returns the total number of files that can be simultaneously opened by your application.

Type: Byte

Value Range: Platform-dependent.

See Also: [Opening Files](#)^[443]

Details

The value of this property depends on the hardware (selected [platform](#)^[133]) and has nothing to do with the formatting of your flash disk. Maximum value of the [fd.filenum](#)^[463] property is fd.maxopenedfiles-1.

.Maxstoredfiles R/O Property

Function:	Returns the total number of files that can be simultaneously stored on the currently existing flash disk.
Type:	Byte
Value Range:	Value depends on the current disk formatting.
See Also:	Creating and Deleting Files ^[441]

Details

This number cannot be changed unless the disk is reformatted using the [fd.format](#)^[467] method.

The disk must be mounted (see [fd.mount](#)^[474]) for this property to return a meaningful value.

.Mount Method

Function:	Mounts the flash disk already existing in the flash memory (prepares it for use).
Syntax:	fd.format() as pl_fd_status_codes
Returns:	One of the following pl_fd_status_codes ^[472] , also affects fd.laststatus ^[472] :
	0- PL_FD_STATUS_OK: Completed successfully. 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal). 2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal). 3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).

See Also:	Mounting the Flash Disk ^[439]
------------------	--

Details

The flash disk will not be accessible unless it is mounted using this method. The disk can only be mounted after the flash memory has been successfully formatted using the [fd.format](#)^[467] method. The disk has to be mounted after every reboot of your device. After the disk is mounted successfully, the [fd.ready](#)^[476] R/O property will read 1- YES.

There is no way to explicitly dismount the disk, nor it is necessary. The disk will be dismounted automatically if any fatal condition is detected when working with the disk. This condition will be reflected by the fd.laststatus R/O property, while the fd.ready will become 0- NO. Note, that not every error indicated by the fd.laststatus is fatal.

The disk will also be dismounted if your application invokes the [fd.format](#)^[467] method or uses the [fd.setsector](#)^[483] to write to a sector belonging to a mounted flash disk (sectors 0 thru [fd.totalsize](#)^{[483]-1}).

.Numservicesectors R/O Property

Function:	Returns the total number of sectors occupied by the "housekeeping" data of the currently existing flash disk.
Type:	Byte
Value Range:	Value depends on the current disk formatting.
See Also:	Checking Disk Vitals ^[441] , Ensuring Disk Data Integrity ^[454] , fd.capacity ^[458] , fd.totalsize ^[483] , fd.format ^[467]

Details

The disk must be mounted (see [fd.mount](#)^[474]) for this property to return a meaningful value.

.Open Method

Function:	Opens a file with a specified name "on" a currently selected file number (selection is made through the fd.filenum ^[463]).
Syntax:	fd.open(byref name as string) as pl_fd_status_codes
Returns:	One of the following pl_fd_status_codes ^[472] , also affects fd.laststatus ^[472] :
	0- PL_FD_STATUS_OK: Completed successfully.
	1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).
	2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).
	3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).
	8- PL_FD_STATUS_NOT_READY: The disk is not mounted.
	9- PL_FD_STATUS_NOT_FOUND: File not found.
	11- PL_FD_STATUS_ALREADY_OPENED: This file is already opened "on" some other file number.

See Also:	Opening Files ^[443]
------------------	--

Par t	Description
----------	-------------

na	A string (1-56 characters) with the file name. All characters after the first space encountered (excluding leading spaces) will be ignored. File names are case-sensitive.
me	

Details

The file string will always be truncated to 56 characters. Leading spaces will be removed. If you open the file "on" the file number which already had another file opened "on" it, the system "forgets" all about the previously opened file. This may potentially lead to the loss of the very latest changes you've made to the previous file. Always close the previous file first by using the [fd.close](#)^[460], or "flush" the changes by using the [fd.flush](#)^[466].

The file with the specified name must exist or the 9- PL_FD_STATUS_NOT_FOUND status codes returned.

The 11- PL_FD_STATUS_ALREADY_OPENED status code is returned if the same file is already opened "on" some other file number.

All other possible error states are "generic" and are not described here.

.Pointer R/O Property

Function:	Returns the pointer position for the file opened "on" the currently selected file number (selection is made through the fd.filenum) ^[463] or zero if no file is currently opened.
Type:	Dword
Value Range:	0 to fd.filesize ^[464] +1 (except for fd.filesize= 0, in which case fd.pointer= 0 too).
See Also:	Writing To and Reading From Files ^[444] , fd.sector ^[478]

Details

For the files of 0 size (see [fd.filesize](#)^[464]), the pointer will always be at 0. If the file has non-zero size, the pointer can be between 1 and fd.filesize+1. "1" is the position of the first byte of the file. The last existing byte of the file is at position equal to the value of fd.filesize. "Fd.filesize+1" is the position at which new data can be added to the file.

The pointer can be moved using the [fd.setpointer](#)^[482] method. Invoking [fd.getdata](#)^[469] or [fd.setdata](#)^[480] moves the pointer forward automatically by the number of bytes that were read from or written to the file.

Reducing the file size through the [fd.setfilesize](#)^[481] may affect the pointer position. If the file size becomes zero, the pointer will also be set to 0. If the new file size is not zero, but its new size makes current pointer position invalid (that is, fd.pointer > fd.filesize+1) then the pointer will be set to fd.filesize+1.

.Ready R/O Property

Function:	Informs whether the flash disk is mounted and ready for use.
Type:	Enum (no_yes, byte)
Value Range:	0- NO: The disk is not mounted and not ready for use (default). 1- YES: The disk is mounted and ready for use.
See Also:	Fd. Object's status codes ^[435] , Mounting the Flash Disk ^[439]

Details

Use the [fd.mount](#)^[474] method to mount this disk.

.Rename Method

Function:	Renames a file specified by its name.
Syntax:	fd.create(byref old_name as string, byref new_name as string) as pl_fd_status_codes
Returns:	One of the following pl_fd_status_codes ^[472] , also affects fd.laststatus ^[472] :
	0- PL_FD_STATUS_OK: Completed successfully. 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal). 2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal). 4- PL_FD_STATUS_INV_PARAM: Old_name is a NULL string, which is not allowed. 5- PL_FD_STATUS_DUPLICATE_NAME: File with the new_name already exists. 8- PL_FD_STATUS_NOT_READY: The disk is not mounted. 9- PL_FD_STATUS_NOT_FOUND: The old_name file is not found.

See Also: [Creating Deleting, and Renaming Files](#)^[441], [File Names and Attributes](#)^[440]

Part Description

old_name	A string (1-56 characters) with the name of the file to be renamed. All characters after the first space encountered (excluding leading spaces) will be ignored. File names are case-sensitive.
new_name	A string (1-56 characters) with the new name for the file. All characters after the first space encountered (excluding leading spaces) will be ignored.

Details

A file can always be renamed, even if this file is currently opened.

The file with the old_name name must be present on the disk, or you will get the 9- PL_FD_STATUS_NOT_FOUND status code. The new_name name must contain at least one character, or 4- PL_FD_STATUS_INV_PARAM code will be returned. The new_name name must be unique, i.e. a file with the same name must not be present on the disk, otherwise the 5- PL_FD_STATUS_DUPLICATE_NAME status code will be generated. Any character except space can be used in file names. This includes "/" and "\". The flash disk does not support subdirectories, but it is possible to emulate them by including "/" or "\"" characters in the file name.

This method will preserve original attributes of the file (use [fd.setattributes](#)⁴⁷⁸ if you want to change those). Since both the filename and attributes must fit into 56 characters (including a space character for separation), renaming a file may lead to a (partial) loss of attribute data. This will happen when the new file name is longer than the old one and the combined length of the new file name and existing attributes (including a separating space) exceeds 56 characters.

.Resetdirpointer Method

Function:	Resets the directory pointer to zero.
Syntax:	fd.resetdirpointer()
Returns:	---
See Also:	Walking Through File Directory ⁴⁴² , fd.getnumfiles ⁴⁷¹

Details

Use this method before repeatedly invoking [fd.getnextdirmember](#)⁴⁷⁰ in order to obtain the list of files currently stored on the disk.

.Sector R/O Property

Function:	Returns the physical sector number corresponding to the current position of the file pointer.
Type:	Word
Value Range:	0-65535
See Also:	Writing To and Reading From Files ⁴⁴⁴ , Ensuring Disk Data Integrity ⁴⁵⁴

Details

Because the sectors belonging to a given file may be scattered around the flash disk, there is no simple way to figure out the number of the physical sector corresponding to the current file pointer position (see [fd.pointer](#)⁴⁷⁶). This property can be used, for instance, to backup a sector prior to making changes to its data.

.Setattributes Method

Function:	Sets the attributes string for a file with the specified file name.
Syntax:	fd.setattributes(byref name as string, byref attr as string) as pl_fd_status_codes

Returns:

One of the following [pl_fd_status_codes](#)^[472], also affects [fd.laststatus](#)^[472]:

- 0- PL_FD_STATUS_OK: Completed successfully.
- 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).
- 2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).
- 3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).
- 8- PL_FD_STATUS_NOT_READY: The disk is not mounted.
- 9- PL_FD_STATUS_NOT_FOUND: File not found.

See Also:

[Reading and Writing File Attributes](#)^[442]

Par t	Description
----------	-------------

na	A string (1-56 characters) with the file name. All characters after the first space encountered (excluding leading spaces) will be ignored. File names are case-sensitive.
me	
attr	A string with attributes to be set. Attributes length cannot exceed 55 characters minus the length of the file name.

Details

The file with the specified name must exist or the 9- PL_FD_STATUS_NOT_FOUND status code will be returned. All other possible error states are "generic" and are not described here.

.Setbuffer Method**Function:**

Writes a specified number of bytes into the currently selected RAM buffer of the flash memory (selection is made through the [fd.buffernum](#)^[458] property).

Syntax:

fd.setbuffer(byref data as string, offset as word) as word

Returns:

Actual number of bytes written. The method also affects the state of the [fd.laststatus](#)^[472]. The following status codes are possible:

- 0- PL_FD_STATUS_OK: Completed successfully.
- 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

See Also:

[Direct Sector Access](#)^[448], [fd.getbuffer](#)^[468], [fd.getsector](#)^[472], [fd.setsector](#)^[483], [fd.checksum](#)^[459]

Par t	Description
----------	-------------

dat A string with the data to written to the buffer.
 a
 offs Starting offset in the buffer. Possible value range is 0-263 (the buffer
 et stores 264 bytes of data, offset is counted from 0).

Details

The length of data actually written into the buffer may be limited if all supplied data can't fit between the offset position in the buffer and the end of the buffer. The method will return the actual number of data that was written.

Note that writing to the RAM buffer #1 may interfere with file-based access to the flash memory. If you have executed [fd.setdata](#)^[480], the RAM buffer #1 may still contain this new data, in which case using the fd.setbuffer while the fd.buffernum=1 will automatically dismount the disk (set [fd.ready](#)^[476]= 0- NO). To avoid this situation, switch to the RAM buffer #0, or use [fd.flush](#)^[466] or [fd.close](#)^[460] methods before invoking the fd.getsector.

All status codes generated by this method are "generic" and are not described here.

.Setdata Method

Function: Writes data to a file opened "on" a currently selected file number (selection is made through the [fd.filenum](#)^[463]).

Syntax: **fd.setdata(byref data as string) as pl_fd_status_codes**

Returns: One of the following [pl_fd_status_codes](#)^[472], also affects [fd.laststatus](#)^[472]:

- 0- PL_FD_STATUS_OK: Completed successfully.
- 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).
- 2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).
- 3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).
- 7- PL_FD_STATUS_DATA_FULL: The disk is full, new data cannot be added.
- 8- PL_FD_STATUS_NOT_READY: The disk is not mounted.
- 10- PL_FD_STATUS_NOT_OPENED: No file is currently opened on the current value of the [fd.filenum](#)^[463] property.

See Also: [Writing To and Reading From Files](#)^[444], [fd.getdata](#)^[469], [fd.sector](#)^[478]

Par	Description
------------	--------------------

dat A string containing data to be written to the file.
 a

Details

As a result of this method invocation, the pointer (see [fd.pointer](#)^[476]) will be advanced forward by the number of bytes written to the file. If the pointer wasn't at the end of the file (at [fd.filesize](#)^[464]+1 position) then (some of) the existing file data will be partially overwritten. If the pointer moves past the current file size (see [filesize](#)^[464]), then the file size will be increased automatically. New data sectors will be allocated and added to the file as needed and provided that there are still free sectors available. 7- PL_FD_STATUS_DATA_FULL status code is returned if the disk runs out of space, in which case the *entire* data string supplied in the fd.setdata is not added to the file (and not just the part that couldn't fit).

All other error codes returned by this method are "generic" and are not described here.

.Setfilesize Method

Function: Sets (reduces) the file size of a file opened "on" a currently selected file number (selection is made through the [fd.filenum](#)^[463]).

Syntax: **fd.setfilesize(newsize as dword) as pl_fd_status_codes**

Returns: One of the following [pl_fd_status_codes](#)^[472], also affects [fd.laststatus](#)^[472]:

- 0- PL_FD_STATUS_OK: Completed successfully.
- 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).
- 2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).
- 3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).
- 8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

See Also: [Removing Data From Files](#)^[445], [fd.cutfromtop](#)^[461]

Par t	Description
------------------	--------------------

ne Desired new file size in bytes. Supplied value will be corrected if wsi exceeded previous file size.
ze

Details

This method cannot be used to increase the size of the file, only decrease it. Data sectors previously allocated to this file will be "released" (marked unused) if they become unnecessary due to the reduction in the file size. The first data sector of the file, however, will always remain allocated, even when the file size is set to 0.

The size of the file (see [fd.filesize](#)^[464]) will be corrected downwards to reflect the amount of data left in the file.

The pointer position (see [fd.pointer](#)^[476]) will be affected by this method. If the file

size becomes empty, the pointer will be set to zero. If the new file size is not zero, but the new size makes current pointer position invalid (that is, fd.pointer > fd.filesize+1) then the pointer will be set to fd.filesize+1.

All possible status codes returned by this method are "generic" and are not described here.

.Setpointer Method

Function:	Sets the new pointer position for a file opened "on" a currently selected file number (selection is made through the fd.filenum ^[463]).
Syntax:	fd.setpointer(pos as dword) as pl_fd_status_codes
Returns:	One of the following pl_fd_status_codes ^[472] , also affects fd.laststatus ^[472] :
	0- PL_FD_STATUS_OK: Completed successfully. 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal). 2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal). 3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal). 8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

See Also: [Writing To and Reading From Files](#)^[444], [fd.sector](#)^[478]

Par	Description
t	

pos Desired new pointer position. Supplied value will be corrected if out of range.

Details

For the files of 0 size (see [fd.filesize](#)^[464]), the pointer may only have one value -- 0. If the file has non-zero size, the pointer can be between 1 and fd.filesize+1. "1" is the position of the first byte of the file. The last existing byte of the file is at position equal to the value of fd.filesize. "Fd.filesize+1" is the position at which new data can be added to the file.

Invoking [fd.getdata](#)^[469] or [fd.setdata](#)^[480] moves the pointer forward automatically by the number of bytes that were read from or written to the file.

Reducing the file size through the [fd.setfilesize](#)^[481] may affect the pointer position. If the file becomes empty, the pointer will be set to 0. If the new file size is not zero, but its new size makes current pointer position invalid (that is, fd.pointer > fd.filesize+1) then the pointer will be set to fd.filesize+1.

Current pointer position can be obtained through the [fd.pointer](#)^[476] R/O property.

All possible status codes returned by this method are "generic" and are not described here.

.Setsector Method

Function:	Writes a specified sector with the data from the currently selected RAM buffer of the flash memory (selection is made through the fd.buffernum ^[458] property).
Syntax:	fd.getsector(num as word) as pl_fd_status_codes
Returns:	One of the following pl_fd_status_codes ^[472] , also affects fd.laststatus ^[472] :
	0- PL_FD_STATUS_OK: Completed successfully. 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

See Also: [Direct Sector Access](#)^[448], [fd.getsector](#)^[472], [fd.getbuffer](#)^[468], [fd.setbuffer](#)^[479], [fd.checksum](#)^[459]

Par	Description
t	

nu Sector to write to. Acceptable range is 0 - [fd.availableflashspace](#)^[457]-1.
m

Details

Each physical sector of the flash memory has 264 bytes of data, and all 264 bytes will be written to.

You can only write to "free" flash sectors that are not occupied by the firmware or Tibbo Basic application of your device. The number of free sectors can be determined through the [fd.availableflashspace](#) R/O property. Trespassing this boundary will result in the 1- PL_FD_STATUS_FAIL status code. This is done to prevent your applications from inadvertently damaging its own code or the firmware. If you want to write into the firmware/application area, use [fd.copyfirmware](#)^[460] instead.

Flash area lying within the [fd.availableflashspace](#) boundary may house a formatted flash disk (see [fd.format](#)^[467], [fd.mount](#)^[474]). Writing to the sector that belongs to a flash disk when the disk is mounted will automatically dismount the disk (set [fd.ready](#)^[476]= 0- NO). **Writing to sectors that belong to the flash disk may corrupt the disk and render it unusable.**

All status codes generated by this method are "generic" and are not described here.

.Totalsize R/O Property

Function:	Returns the total number of sectors occupied by the currently existing flash disk.
Type:	Word
Value Range:	0-65535
See Also:	Checking Disk Vitals ^[441] , fd.availableflashspace ^[457]

Details

For internal housekeeping reasons, the total size of the disk as returned by this property may be less than the size that was requested when formatting the disk with the [fd.format](#)^[467] method. Actual usable capacity ([fd.capacity](#)^[458]) of the disk is less because the disk also needs a number of sectors for its "housekeeping" data (see [fd.numservicesectors](#)^[475]).

The disk must be mounted (see [fd.mount](#)^[474]) for this property to return a meaningful value.

Kp Object



This is the keypad (kp.) object, it allows you to work with a "matrix" keypad of up to 64 keys (8 scan lines by 8 return lines).

Features:

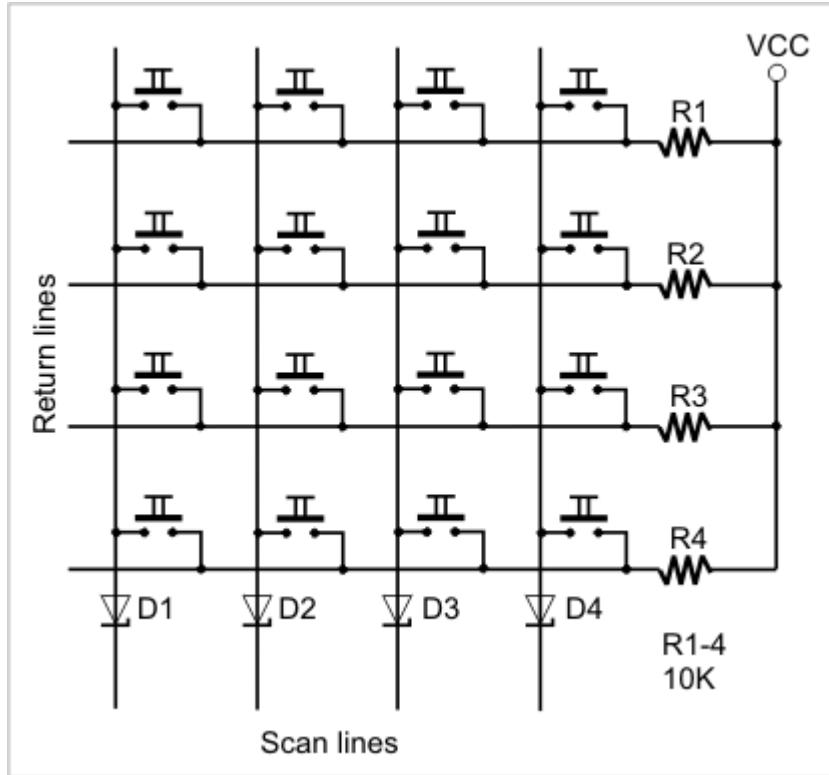
- Flexible [keypad arrangement](#)^[484]. Scan lines can double as LED control lines too.
- Five [distinctive states](#)^[486] for each key allow you to create sophisticated keypad input. Individual programming of [delay times](#)^[487] for each state transition.
- Flexible [mapping](#)^[487] for scan and return lines -- use any I/O lines, in any order.
- Ability to [auto-disable](#)^[487] the keypad when a certain key event/code combination is encountered.

Possible Keypad Configurations

The kp object works with a "matrix" keypad formed by scan and return lines. The keypad object supports up to 8 scan and 8 return lines, which means that you can build a keypad with up to 64 keys. A sample schematic diagram for a typical keypad is shown below.

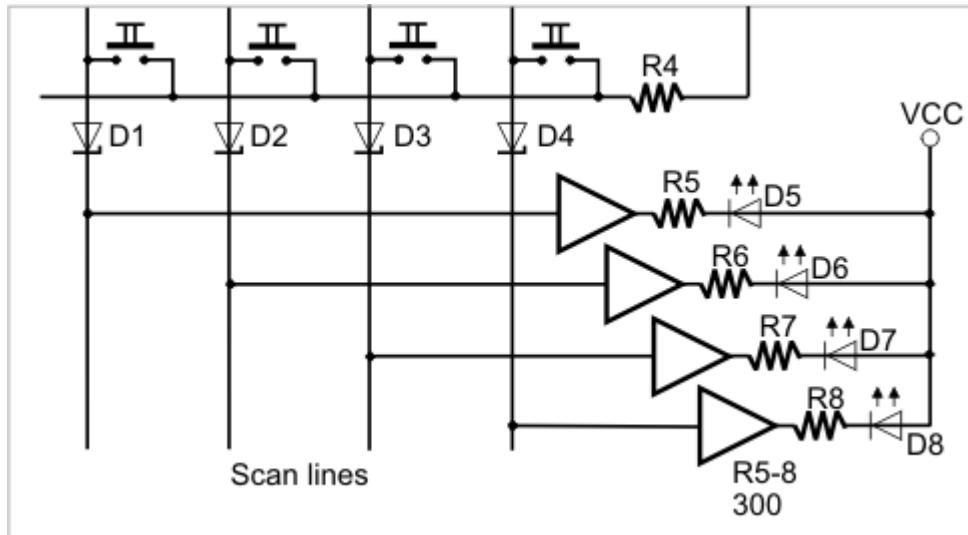
During the scanning process, the kp object "activates" one scan line after another. The line is activated by setting it LOW, while keeping all other scan lines HIGH. For each scan line, the kp object samples the state of return lines. If any return line is at LOW, this means the key located at the intersection of this return line and the currently active scan line is pressed.

A detailed discussion of the schematics falls outside the scope of this manual. We will only notice, in passing, that the diodes D1-D4 are necessary and should not be omitted. For best result, use Schottky diodes -- they have low drop voltage. Pull-up resistors R1-R4 prevent the return lines from floating and should be present as well.



On platforms with [output buffer control](#)^[368], all intended scan lines should be configured as outputs, and all return lines -- as inputs (see [io.num](#)^[372], [io.enabled](#)^[370]).

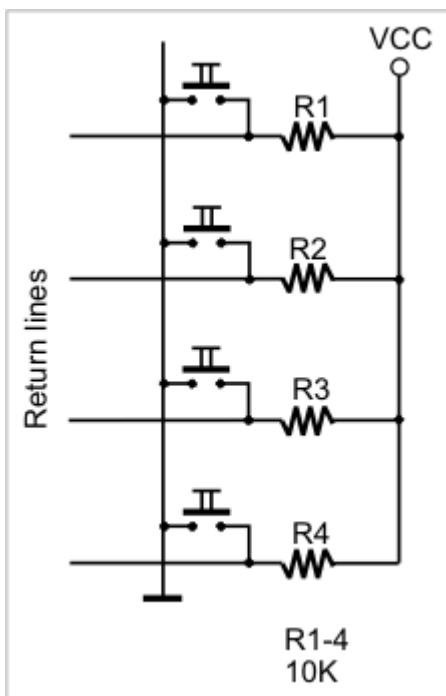
Scan lines can optionally perform a double duty and drive LEDs. One such LED can be connected to each scan line, preferably through a buffer, as shown on the drawing below. These LEDs can be used for any purpose you desire -- and this purpose can be completely unrelated to the keypad itself.



If the LED is connected as shown on the drawing, you need to set the corresponding I/O line LOW in order to turn this LED on. Each time the kp. object is to scan the keypad for pressed keys -- and this happens every 10ms -- it will

first set all scan lines to HIGH. This is necessary for correct keypad operation. Before doing so, however, the kp object will memorize the state of each scan line. This state will be restored after the scanning is complete. To your eye, this will look like that LED connected to the scan line was on all the time (of course, it it was on in the first place).

To build a functioning keypad you will need to have at least one return line. A sensible count of scan lines, however, starts from two! Having a single scan line is like having no scan lines whatsoever -- you might just as well ground this single scan line of the keypad, i.e. keep it active permanently. This arrangement is shown on the drawing below.



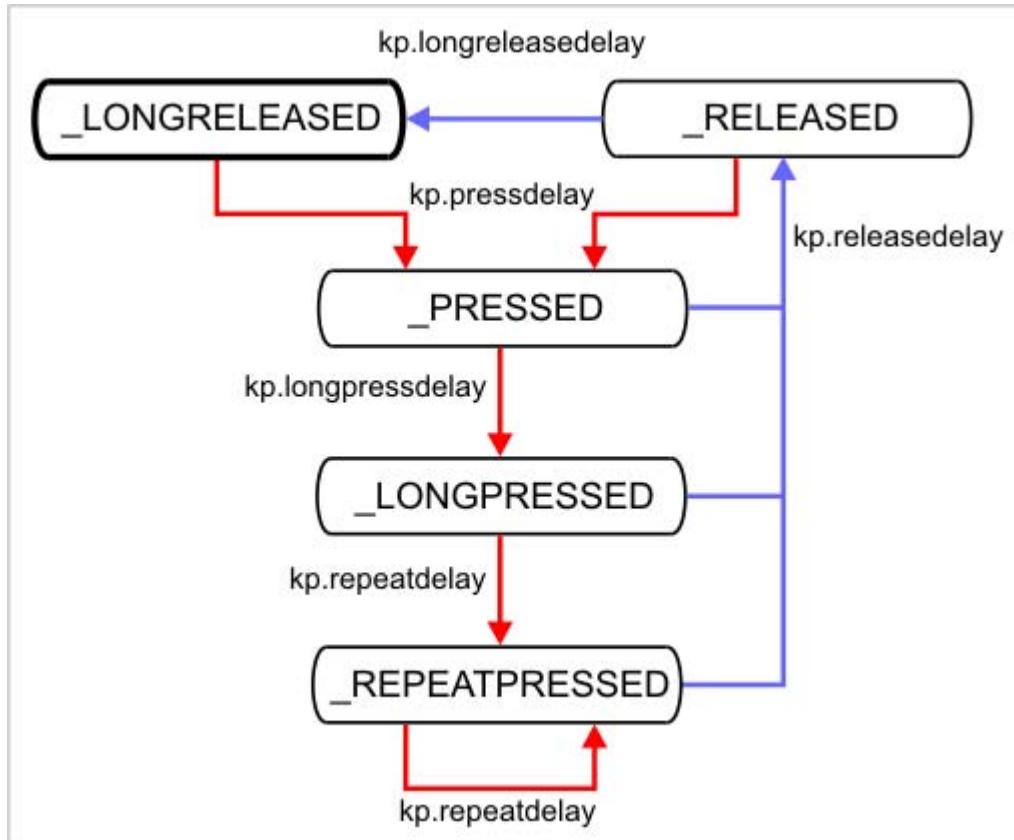
[Preparing the Keypad for Operation](#) topic explains how to setup the kp. object properly for the keypad hardware you are using.

Key States and Transitions

Each key on your keypad can be in five different states, as defined by the pl_kp_event_codes constant. Here are those states:

- 0- PL_KP_EVENT_LONGRELEASED: The key has been released "for a while".
- 1- PL_KP_EVENT_RELEASED: The key has been released (just now).
- 2- PL_KP_EVENT_PRESSED: The key has been pressed (just now).
- 3- PL_KP_EVENT_LONGPRESSED: The key has been pressed "for a while".
- 4- PL_KP_EVENT_REPEATPRESSED: Auto-repeat for the key.

The diagram below shows all key states and possible state transitions.



Possible state transitions are indicated by arrows. Red arrows show transitions for when the key is pressed (or remains pressed), blue -- for when the key is released (or remains released). The time it takes for the key to transition from one state to another is quantified in 10ms intervals. This is the time period at which the kp. object will perform keypad scans. Each transition delay, expressed in 10ms intervals, is defined by a dedicated property, so you can decide for yourself what "just now" and "for a while" mean. Each time the key transitions to the next state, the [on_kp](#)^[491] event is generated. See [Servicing Keypad Events](#)^[489] for how to work with events. The keypad object has a buffer that can hold up to 8 keypad events.

When the keypad is enabled ([kp.enabled](#)^[492] is set to 1- YES), the keypad buffer is cleared and each key's state is set to "longreleased". Press the key long enough, and the key will go into the "pressed" state. From there, the key can go to "released" and, later, "longreleased" if you let go of that key, or into "longpressed" and then "repeatpressed" if you keep the key pressed.

Five available key states, along with adjustable state transition times, allow you to create sophisticated keypad input. For example, it is possible to have a mixed alphanumerical input, like the one used on mobile phones for SMS entry.

"Longpressed" events can be assigned to add a digit, for example, "1" if you press the "1ABC" key. "Pressed" events rotate between letters of the key ("A"->"B"->"C"->"A", etc.), unless another key is pressed or rotation times out on "longreleased" event. In both cases, input advances to the next character.

Preparing the Keypad for Operation

This topic explains what you need to do to properly set up the kp. object. All preparations should be made with the keypad disabled ([kp.enabled](#)^[492]= 0- NO), or this setup won't work.

Mapping scan and return lines

First, you need to define the list of scan and return lines. One great feature of the kp. object is that you can assign ("map") any I/O line of your device to be a scan or return line. I/O lines serving as scan or return lines don't even have to be "together" (have consecutive numbers). Scan lines are assigned through the [kp.scanlinesmapping](#)⁴⁹⁷ property, return lines -- through the [kp.returnlinesmapping](#)⁴⁹⁸ property. For example, here is how you select lines 24, 20, and 27 to serve as scan lines, and 28, 21, and 25 to serve as return lines:

```
kp.scanlinesmapping="24,20,27"
kp.returnlinesmapping="28,21,25"

io.num=PL_IO_NUM_24
io.enabled=YES
io.num=PL_IO_NUM_20_INT4
io.enabled=YES
io.num=PL_IO_NUM_27
io.enabled=YES
```

On platforms with [output buffer control](#)³⁶⁸ each scan line must be configured as output, each return line -- as input (shown in the example above).

Notice how we did not have any order in specifying return and scan lines -- this simply does not matter. Select any lines, order them in any way you want. The only limitations are:

- You can't have more than 8 scan lines and 8 return lines;
- You must have at least 1 return line;
- Any given I/O line can only serve as a scan or return line, not both.

Line numbers are platform-dependent. They come from pl_io_num found in the "Platform-dependent Constants" section of your platform documentation. The pl_io_num is a list of constants like "PL_IO_NUM_24". You cannot just drop "PL_IO_NUM_24" into kp.scanlinesmapping (or kp.returnlinesmapping). The correct way is to write "24" or use str(PL_IO_NUM_24). So, another way to do the above setup would look like this:

```
kp.scanlinesmapping=str(PL_IO_NUM_24)+","+str(PL_IO_NUM_20_INT4)+","+str(PL_IO_NUM_27)
kp.returnlinesmapping=str(PL_IO_NUM_28)+","+str(PL_IO_NUM_21_INT5)+","+str(PL_IO_NUM_25)
...
s=kp.returnlinesmapping 's will be equal to '28,21,25' (full constant
names like PL_IO_NUM_28 are not preserved)
```

Notice that no matter how you set kp.scanlinesmapping and kp.returnlinesmapping, reading them will always return a simple list of numbers (shown in the example above).

Defining state transition delays

Your second step is to set proper delay times for [key state transitions](#)⁴⁸⁶. Five

different properties are responsible for that: [kp.pressdelay](#)^[495], [kp.longpressdelay](#)^[495], [kp.repeatdelay](#)^[495], [kp.releasedelay](#)^[495], and [kp.longreleasedelay](#)^[495]. Each property sets the transition delay time in 10ms increments. Setting a property to 0 means that the corresponding transition will never happen. Note that the maximum value for each property is 254, not 255. All five properties already have sensible default values, so you only need to change them if you don't like what we have chosen for you:

```
kp.pressdelay=4 '40ms (4 successive keypad scans) to confirm that the key
is pressed
kp.longpressdelay=150 '1.5 seconds
kp.repeatdelay=0 'we do not want auto-repeat to work
kp.releasedelay=4 '40ms
kp.longreleasedelay=200 '2 seconds
```

Keypad auto-disable

Finally, you can select several event/code combinations that will automatically disable the keypad. This is done through the [kp.autodisablecodes](#)^[492] property. Each time one of the pre-sent combinations of the key state and key code is detected, the kp.enabled property will be set to 0- NO, thus preventing further input until you re-enable the keypad.

This behavior can be very useful. Supposing, you have an application where you need to enter a certain code, then press <ENTER>. After you press <ENTER>, your application processes the input, which may take some time. What you often need is to prevent any further keypad input while the code is being processed. If you do not do this, the user might continue punching away and creating garbage input that your system does not need. This might even overwhelm the keypad buffer (see [Servicing Keypad Events](#)^[489]). The kp.autodisablecodes makes sure that the input stops at a certain event/code combination. Up to four such combinations can be defined.

Here is an example of how this property could be set:

```
kp.autodisablecodes=str(PL_KP_EVENT_PRESSED)+",49" 'assuming that <ENTER>
key has the code of 49
kp.autodisablecodes="2,49" 'this is because the PL_KP_EVENT_PRESSED
constant is equal to 2
```

Once all the properties have been preset, enable the kp. object (kp.enabled= 1- YES), and start processing [keypad events](#)^[489].

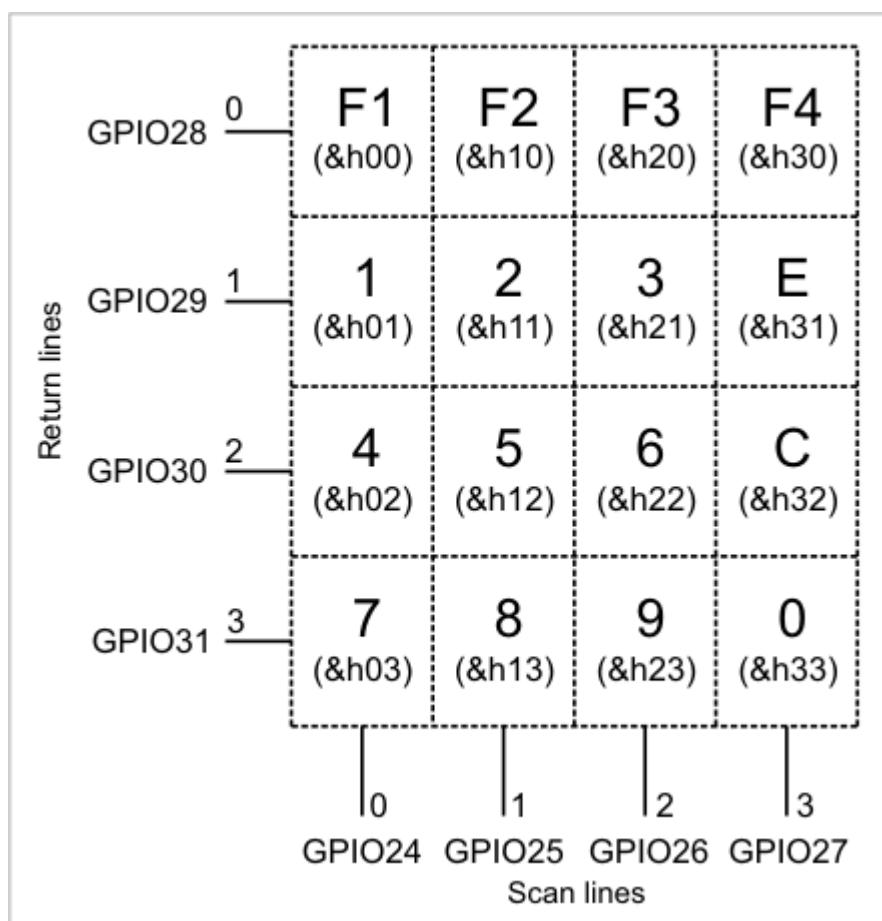
Servicing Keypad Events

Once you have correctly [preset and enabled](#)^[487] the kp. object, you only need to process keypad events.

The [on_kp](#)^[494] is the main event that is generated each time a key transitions to a new [key state](#)^[486]. The key_event argument will tell you what that new state is, while the key_code will tell you the code of the key. Event codes are defined in the

on_kp_event_codes enum. The key code is composed of the scan line number (bits 7-4 of the key code), and the return line number (bits 3-0). Scan and return lines are numbered in the same order they are listed in the [kp.scanlinesmapping](#)^[497] and [kp.returnlinesmapping](#)^[498] properties.

For example, supposing you have a 4x4 keypad with <0> - <9> keys, also <F1> - <F4>, <E> (enter) and <C> (cancel). Notice how key codes in their hex representation reflect the number of the scan line (high digit) and return line (low digit):



Set up the kp. object to work correctly with your hardware:

```

kp.scanlinesmapping="24,25,26,27"
kp.returnlinesmapping="28,29,30,31"

io.num=PL_IO_NUM_24
io.enabled=YES
io.num=PL_IO_NUM_25
io.enabled=YES
io.num=PL_IO_NUM_26
io.enabled=YES
io.num=PL_IO_NUM_27
io.enabled=YES

'we are not going to change default delay values -- we like them as they
are (we came up with them, after all)

kp.autodisablecodes=str(PL_KP_EVENT_PRESSED)+",49" '<ENTER> will disable
further input

kp.enabled=YES

```

Here is an example of the event handler that adds your input to the inp_str string (global variable), clears the string when the <CANCEL> key is pressed, and launches the mysterious process_it procedure when the <ENTER> key is pressed:

```

Sub On_kp(key_event As pl_kp_event_codes, key_code As Byte)
    Dim x As Byte
    If key_event=PL_KP_EVENT_PRESSED Then
        Select Case key_code
            Case &h1: x=1
            Case &h11: x=2
            Case &h21: x=3
            Case &h2: x=4
            Case &h12: x=5
            Case &h22: x=6
            Case &h3: x=7
            Case &h13: x=8
            Case &h23: x=9
            Case &h33: x=0
            Case &h32: '<CANCEL>
                inp_str=""
                Exit Sub
            Case &h31: '<ENTER> will disable the keypad...
                process_it()
                inp_str=""
                kp.enabled=YES '... so we re-enable it
                Exit Sub
        End Select
        inp_str=inp_str+chr(x) 'we will be here only when a numerical key is
        pressed (see 'exit sub' under CANCEL and ENTER keys)
    End If
End Sub

```

Handling keypad buffer overflows

Another event -- [on kp overflow](#)⁴⁹⁴ -- tells you that the input buffer of the keypad has been overwhelmed with frantic user input and the kp. object is not disabled. You respond appropriately:

```

Sub On_kp_overflow
    Lcd.print("WHAT'S THE RUSH? SLOW DOWN!",0,0) 'tell the user
    inp_str="" 'clear the string
    kp.enabled=YES 're-enable the keypad
End Sub

```

Properties, Methods, Events

Properties, methods, and events of the kp object.

.Autodisablecodes Property

Function:	Defines which key event/code combinations disable the keypad.
Type:	String
Value Range:	Up to four comma-separated event/code pairs. Default= "".
See Also:	Preparing the Keypad for Operation ^[487] , Key States and Transitions ^[486]

Details

This property should contain a comma-separated list of event codes and key codes, for example: "2,15,0,20". In this example, two event/code pairs are set: "2,15" and "0,20". Event "2" is 2- PL_KP_EVENT_PRESSED, and event "0" is 0-PL_KP_EVENT_LONGRELEASED (see [on_kp](#)^[494] event for a full list of codes). "15" and "20" are key codes. So, the keypad will be disabled ([kp.enabled](#)^[492] set to 0-NO) when the key with code 15 is detected to be "pressed", or the key with code 20 is detected to be "longreleased".

The kp.autodisablecodes string should only contain a list of decimal numbers. That is, use "2" and not "2- PL_KP_EVENT_PRESSED". Only numerical characters are processed anyway -- writing "2- PL_KP_EVENT_PRESSED,15,0-PL_KP_EVENT_LONGRELEASED,20" will set this property to "2,15,0,20" anyway. You can, of course, write str(PL_KP_EVENT_PRESSED)+","+"15"+","+str(PL_KP_EVENT_LONGRELEASED)+","+"20" instead of "2,15,0,20".

This property can only be changed when the keypad is disabled ([kp.enabled](#)=0-NO). Setting the property to "" means that no event and key combination will disable the keypad automatically.

.Enabled Property

Function:	Enables or disables the keypad.
Type:	Enum (no_yes, byte)
Value Range:	0- NO (default): The keypad is disabled. 1- YES: The keypad is enabled.
See Also:	Preparing the Keypad for Operation ^[487]

Details

The keypad matrix is being scanned and your application receives the [on_kp](#)⁴⁹⁴ and [on_kp_overflow](#)⁴⁹⁴ events only when the keypad is enabled (`kp.enabled= 1- YES`).

The following properties can be changed only when the keypad is disabled (`kp.enabled= 0- NO`): [kp.autodisablecodes](#)⁴⁹², [kp.longreleasedelay](#)⁴⁹³, [kp.longpressdelay](#)⁴⁹³, [kp.pressdelay](#)⁴⁹⁵, [kp.repeatdelay](#)⁴⁹⁶, [kp.returnlinesmapping](#)⁴⁹⁶, [kp.scanlinesmapping](#)⁴⁹⁷.

The keypad will be auto-disabled if an overflow is detected (see [on_kp_overflow](#)⁴⁹⁴ event), or if one of the conditions for automatic keypad disablement is met (see [kp.autodisablecodes](#)⁴⁹²).

Every time the keypad is enabled, each key's state is set to 0-PL_KP_EVENT_LONGRELEASED and the keypad event buffer is cleared.

.Longpressdelay Property

Function:	Defines (in 10ms increments) the amount of time a key should remain pressed for the key state to transition from "pressed" into "longpressed".
Type:	Byte
Value Range:	0-254. Default = 100 (1000ms).
See Also:	Key States and Transitions ⁴⁸⁶ , Preparing the Keypad for Operation ⁴⁸⁷

Details

The [on_kp](#)⁴⁹⁴ event with 3- PL_KP_EVENT_LONGPRESSED event code will be generated once the key transitions into the "longpressed" state.

This property can only be changed when the keypad is disabled ([kp.enabled](#)⁴⁹²= 0-NO). Setting the property to 0 means that the key will never transition into the "longpressed" state.

.Longreleasedelay Property

Function:	Defines (in 10ms increments) the amount of time a key should remain released for the key state to transition from "released" into "longreleased".
Type:	Byte
Value Range:	0-254. Default = 100 (1000ms).
See Also:	Key States and Transitions ⁴⁸⁶ , Preparing the Keypad for Operation ⁴⁸⁷

Details

The [on_kp](#)⁴⁹⁴ event with 0- PL_KP_EVENT_LONGRELEASED event code will be

generated once the key transitions into the "longreleased" state.

This property can only be changed when the keypad is disabled ([kp.enabled](#)^[492]= 0-NO). Setting the property to 0 means that the key will never transition into the "longreleased" state.

On_kp Event

Function:	Generated whenever a key transitions to another state.
Declaration:	on_kp(key_event as pl_kp_event_codes, key_code as byte)
See Also:	Servicing Keypad Events ^[489] , on_kp_overflow ^[494]

Part	Description
key_event	0- PL_KP_EVENT_LONGRELEASED: The key has transitioned into the "longreleased" state. 1- PL_KP_EVENT_RELEASED: The key has transitioned into the "released" state. 2- PL_KP_EVENT_PRESSED: The key has transitioned into the "pressed" state. 3- PL_KP_EVENT_LONGPRESSED: The key has transitioned into the "longpressed" state. 4- PL_KP_EVENT_REPEATPRESSED: Auto-repeat for the key.
key_code	Key code (byte). Bits 7-4 of this code represent scan line number, bits 3-0 -- return line number.

Details

Pressing and releasing any key on the keypad can generate up to five different events, as explained in [Key States and Transitions](#)^[486]. Scan lines and return lines are numbered as they are listed in [kp.returnlinesmapping](#)^[496] and [kp.scanlinesmapping](#)^[497].

This event can only be generated when the keypad is enabled ([kp.enabled](#)^[492]= 1-YES).

On_kp_overflow Event

Function:	Indicates that the keypad buffer has overflowed and some key events may have been lost.
Declaration:	on_kp_overflow
See Also:	Servicing Keypad Events ^[489]

Details

The keypad buffer stores up to 16 keypad events. Each such event causes the [on_kp](#)^[494] generation. If your application is slow to process the keypad events, it is possible to overflow the keypad by pressing the keys in rapid succession. Once the buffer overflows, the keypad is disabled automatically ([kp.enabled](#)^[492] is set to 0-NO). You can re-enable the keypad by setting kp.enabled= 1- YES (this will clear the keypad buffer).

.Pressdelay Property

Function:	Defines (in 10ms increments) the amount of time a key should remain pressed for the key state to transition from "released" into "pressed".
Type:	Byte
Value Range:	0-254. Default = 3 (30ms).
See Also:	Key States and Transitions ^[486] , Preparing the Keypad for Operation ^[487]

Details

The [on_kp](#)^[494] event with 2- PL_KP_EVENT_PRESSED event code will be generated once the key transitions into the "pressed" state.

This property can only be changed when the keypad is disabled ([kp.enabled](#)^[492]= 0-NO). Setting the property to 0 means that the key will never transition into the "pressed" state.

.Releasedelay Property

Function:	Defines (in 10ms increments) the amount of time a key should remain released for the key state to transition from "pressed" or "longpressed" into "released".
Type:	Byte
Value Range:	0-254. Default = 3 (30ms).
See Also:	Key States and Transitions ^[486] , Preparing the Keypad for Operation ^[487]

Details

The [on_kp](#)^[494] event with 1- PL_KP_EVENT_RELEASED event code will be generated once the key transitions into the "released" state.

This property can only be changed when the keypad is disabled ([kp.enabled](#)^[492]= 0-NO). Setting the property to 0 means that the key will never transition into the "released" state.

.Repeatdelay Property

Function:	Defines (in 10ms increments) the time period at which the on_kp ^[494] event with 4-PL_KP_EVENT_REPEATPRESSED event code will be generated once the key reaches the "longpressed" state and remains pressed.
Type:	Byte
Value Range:	0-254. Default = 50 (500ms).
See Also:	Key States and Transitions ^[486] , Preparing the Keypad for Operation ^[487]

Details

This property can only be changed when the keypad is disabled ([kp.enabled](#)^[492]= 0-NO). Setting the property to 0 means that the on_kp event with 4-PL_KP_EVENT_REPEATPRESSED event code will never be generated.

.Returnlinesmapping Property

Function:	Defines the list of up to 8 I/O lines that will serve as return lines of the keypad matrix.
Type:	String
Value Range:	Up to eight comma-separated I/O line numbers can be listed. Default = "".
See Also:	Possible Keypad Configurations ^[484] , Preparing the Keypad for Operation ^[487] , kp.scanlinesmapping ^[497]

Details

This property should contain a comma-separated list of I/O lines numbers, for example: "24, 26, 27". Line numbers correspond to those of the pl_io_num enum. This enum is platform-specific. The declarations for the pl_io_num can be found in the "Platform-dependent Constants" section of your platform documentation.

The kp.returnlinesmapping string should only contain a list of decimal numbers. That is, use "24" and not "24- PL_IO_NUM_24". Only numerical characters are processed anyway -- writing "24- PL_IO_NUM_24,25-PL_IO_NUM_25" will set this property to "24,25". You can, of course, write str(PL_IO_NUM_24)+","+str(PL_IO_NUM_25) as well.

The order in which you list the return lines *does matter* -- this is the order in which the lines will be numbered. All keys connected to the first return line will have their return field (bits 4-0) of the key code set to 0. For keys connected to the second line this field will contain 1, third line -- 2, and so on.

On platforms with [output buffer control](#)^[368], all intended return lines should be configured as inputs by your application (see [io.num](#)^[372], [io.enabled](#)^[370]).

This property can only be changed when the keypad is disabled ([kp.enabled](#)^[492]= 0-NO). Setting the property to "" means that the keypad will have no return lines. A

keypad must have at least one return line to be able to work.

Return lines of the keypad should be separate from the scan lines (see [kp.scanlinesmapping](#)^[497]). The keypad will not work properly if you designate any I/O line as both a scan and return line.

.Scanlinesmapping Property

Function:	Defines the list of up to 8 I/O lines that will serve as scan lines of the keypad matrix.
Type:	String
Value Range:	Up to eight comma-separated I/O line numbers can be listed. Default = "".
See Also:	Possible Keypad Configurations ^[484] , Preparing the Keypad for Operation ^[487] , kp.scanlinesmapping ^[497]

Details

This property should contain a comma-separated list of I/O lines numbers, for example: "28, 30, 31". Line numbers correspond to those of the pl_io_num enum. This enum is platform-specific. The declarations for the pl_io_num can be found in the "Platform-dependent Constants" section of your platform documentation.

The kp.scanlinesmapping string should only contain a list of decimal numbers. That is, use "28" and not "28- PL_IO_NUM_28". Only numerical characters are processed anyway -- writing "28- PL_IO_NUM_28,30-PL_IO_NUM_30" will set this property to "28,30". You can, of course, write str(28- PL_IO_NUM_28)+","+str(30- PL_IO_NUM_30) as well.

The order in which you list the scan lines *does matter* -- this is the order in which the lines will be numbered. All keys connected to the first scan line will have their scan field (bits 7-4) of the key code set to 0. For keys connected to the second line this field will contain 1, third line -- 2, and so on.

On platforms with [output buffer control](#)^[368], all intended scan lines should be configured as outputs by your application (see [io.num](#)^[372], [io.enabled](#)^[370]).

This property can only be changed when the keypad is disabled ([kp.enabled](#)^[492]= 0-NO). Setting the property to "" means that the keypad will have no scan lines, which is also a *valid* keypad configuration.

Scan lines of the keypad should be separate from the return lines (see [kp.returnlinesmapping](#)^[496]). The keypad will not work properly if you designate any I/O line as both the scan line and return line.

Wln Object



The wln object represents the Wi-Fi interface of your device. It is through this object that you find available Wi-Fi networks and select the one to associate with. You can also create an ad-hoc network of your own and have other stations

connect to it.

The wln object is not responsible for actual data communications over the Wi-Fi -- this is the job of the [sock](#)^[274] object. In the task it performs, the wln object is similar to the [net](#)^[267] object, which controls another interface -- the Ethernet. In comparison, the wln object is more complex.

The wln object allows you to:

- Scan for available networks and obtain their parameters such as name, channel, mode, etc.
- Set WEP security mode and key.
- Associate with one of the networks (at a time) or form your own "ad-hoc" network on a desired channel.
- Monitor received signal strength.
- Detect disassociation from the network.
- Detect Wi-Fi interface power-down or malfunction.

The present incarnation of the wln object is intended to work with Tibbo's GA1000 802.11b/g add-on board. Previous version of the .wln object worked with the now obsolete WA1000 add-on board. This transition has caused certain changes in how the wln object works. These changes are detailed in [Migrating From the WA1000](#)^[498] topic.

Migrating From the WA1000

This topic details the differences between the previous and current incarnation of the wln object. This difference is related to switching to the new GA1000 hardware. The old WA1000 Wi-Fi add-on module has been replaced with the new GA1000 device. The wln object has been affected, but the changes are minimal. Here is what's different:

- [Wln Tasks](#)^[500]: two new tasks -- [wln.networkstart](#)^[521] and [wln.networkstop](#)^[522].
- [Wln State Transitions](#)^[503]: now includes [creating](#)^[512] and [terminating](#)^[512] own ad-hoc network.
- [Applying Reset](#)^[506]: you can't turn the power on/off anymore, you can just reset the Wi-Fi;
- [Configuring Interface Lines](#)^[507]: all interface lines can now be remapped, not just the CS line. Any 5 I/O lines (including RST) of your programmable module can be used to control the Wi-Fi add-on. See [wln.clkmap](#)^[517], [wln.dimap](#)^[517], [wln.domap](#)^[518].

- [Setting MAC address](#)^[507] ([wln.mac](#)^[520]): it is now optional. The Wi-Fi add-on has a factory-assigned MAC. Leave the wln.mac uninitialized ("0.0.0.0.0.0"), boot up ([wln.boot](#)^[515]) and the pre-assigned MAC will be used.
- [Selecting Domain](#)^[508]: domain list has changed -- see [wln.domain](#)^[518].
- [Booting Up the Hardware](#)^[508]: the firmware file for the Wi-Fi add-on is now called "ga1000fw.bin".
- [Setting TX Power](#)^[509]: power selection is in the 4-15 range now (see [wln.settxpower](#)^[527]).
- Setting WEP Mode and Key: no more [wln.wepkey1](#)^[529]...[wln.wepkey4](#)^[529], [wln.wepmode](#)^[530] deleted too. The [wln.setwep](#)^[527] method now accepts the mode and the key directly. Keys 2-4 are not used, so there is only one key to set.
- [Associating With Selected Network](#)^[511]: [wln.associate](#)^[513] now accepts bssmode and ssid as arguments, so [wln.bssmode](#)^[515] and [wln.ssid](#)^[528] have been deleted. Also, the wln.associate won't cause the Wi-Fi interface to start its own ad-hoc network. This is now done using [wln.networkstart](#)^[521].
- [Creating Own Ad-hoc Network](#)^[512] ([wln.networkstart](#)^[521]), [Terminating Own Ad-hoc Network](#)^[512] ([wln.networkstop](#)^[522]) are completely new topics. [wln.associationstate](#)^[514] now has a new state: 2- PL_WLN_OWN_NETWORK. [Wln.defaultibsschannel](#)^[517] is gone -- wln.networkstart takes this as argument.

Overview 3.15.2

If you are new to Wi-Fi communications, then we recommend you to read the [Wi-Fi Parlance Primer](#)^[500] that will introduce you to some important Wi-Fi lingo.

[Wln Tasks](#)^[500] topic explains the basics of interaction with the wln object.

The rest of the manual follows the natural sequence of steps that you usually take when working with and through the Wi-Fi interface ([Wln State Transitions](#)^[503] topic expands on the subject). The steps are as follows:

- [Brining up Wi-Fi interface](#)^[504]
- [Scanning for Wi-Fi networks](#)^[509]
- [Setting WEP mode and key](#)^[510]
- [Associating with selected network](#)^[511] (or [creating own ad-hoc network](#)^[512])
- [Communicating via Wi-Fi interface](#)^[512]
- [Disassociating from the network](#)^[512] (or [terminating own ad-hoc network](#)^[512])
- [Rebooting](#)^[513]
- [Detecting disassociation or offline state](#)^[513]

Wi-Fi Parlance Primer

If you are new to Wi-Fi networking, this section will provide you with a bit of knowledge on the highly specialized jargon used in the Wi-Fi world. We also provided useful links that will lead you to a lot more info on the subject.

Here are the abbreviations and terms that we will use:

- **BSS** stands for Basic Service Set (ah, now it is clear!). In simple words, this is a wireless network created by a single access point (http://en.wikipedia.org/wiki/Wireless_access_point), but not always. It all depends on the...
- **BSS mode**. Wi-Fi network can operate in either the infrastructure mode, or ad-hoc mode. In the infrastructure mode, a wireless access point is used to create, control and regulate the network. In the ad-hoc mode, there is no access point and wireless devices communicate directly to each other. This is called "IBSS" (Independent Basic Service Set), more on this here: http://en.wikipedia.org/wiki/Independent_Basic_Service_Set.
- **SSID** stands for Service Set Identifier. In simple words, the SSID is a name of the wireless network. In case of the infrastructure network, this name is preset on the access point during its configuration. You can read more on SSIDs here: <http://en.wikipedia.org/wiki/SSID>.
- **BSSID** stands for Basic Service Set Identifier. This is the "MAC address of the wireless network". When your network is built on the access point, this is the MAC address of this access point. For ad-hoc networks the BSSID is selected with certain randomness built in. More on this here: http://en.wikipedia.org/wiki/BSSID#Basic_service_set_identifier.
- **Channel**. Wi-Fi devices operate on one of 14 preset frequencies. Channel refers to the channel number, not the actual frequency used. Depending on the locale, you can be restricted to fewer channels: http://en.wikipedia.org/wiki/List_of_WLAN_channels.
- **RSSI**. Stands for Received Signal Strength Indication. This is a measure of the quality of RF signal received from the wireless network (or peer). More on this here: <http://en.wikipedia.org/wiki/Rssi>.
- **WEP** stands for Wired Equivalent Privacy, a widely used method of protecting Wi-Fi networks from eavesdropping and unauthorized access. The name carries a bit of a wishful thinking, as it has been clearly demonstrated that WEP is rather weak and can be defeated. Read on here: http://en.wikipedia.org/wiki/Wired_Equivalent_Privacy.

Wln Tasks

True to the [non-blocking operation philosophy](#)⁴ of the entire system, the wln object does not stall the entire Tibbo Basic application execution to wait for the wln interface to complete required operation ("task"). You program gives the wln object a task to perform, and then it is free to go and do other things.

There are seven wln tasks:

- [Setting TX power](#)⁵⁰⁸ (initiated by [wln.settxpower](#)⁵²⁷ method).
- [Scanning for Wi-Fi networks](#)⁵⁰⁹ (initiated by [wln.scan](#)⁵²⁴ method).
- [Setting WEP mode and key](#)⁵¹⁰ (initiated by [wln.setwep](#)⁵²⁷ method).
- [Associating with selected network](#)⁵¹¹ (initiated [wln.associate](#)⁵¹³ by method).
- [Creating own ad-hoc network](#)⁵¹² (initiated by [wln.networkstart](#)⁵²¹ method).
- [Disassociating from the network](#)⁵¹² (initiated by [wln.disassociate](#)⁵¹⁸ method).
- [Terminating own ad-hoc network](#)⁵¹² (initiated by [wln.networkstop](#)⁵²² method).

The wln object can only work on a single task at a time. You cannot request another task execution until the previous one is finished! The following example shows a wrong way of tasking:

```
'THIS WON'T WORK!
...
wln.scan("NET1")
wln.associate(wln.scanresultbssid,"NET1",wln.scanresultchannel,wln.scanres
ultbssmode) 'this task will be skipped over!
```

Here is how you should do this: use the [wln.task](#)⁵²⁸ read-only property and wait until the previous task is completed.

```
'A BETTER WAY...
...
While wln.task<>PL_WLN_TASK_IDLE 'waiting for the previous task to
complete...
Wend
wln.scan("NET1")
...
While wln.task<>PL_WLN_TASK_IDLE 'waiting for the previous task to
complete...
Wend
wln.associate(wln.scanresultbssid,"NET1",wln.scanresultchannel,wln.scanres
ultbssmode)
...
```

The above approach still needs some refinement. Just making sure that the previous task has completed will not guarantee that your next task will be accepted. This is because some tasks can only be accepted under certain additional conditions. For example, you can't scan while being associated with a wireless network (or running an ad-hoc network of your own). Try this, and the [wln.scan](#)⁵²⁴ will return 1- REJECTED.

```
'A MUCH BETTER WAY...
...
While wln.task<>PL_WLN_TASK_IDLE 'waiting for the previous task to
complete...
Wend
If wln.scan("NET1")<>ACCEPTED Then
    'Handle this: there must be a reason why the task got rejected.
    'We already made sure that the previous task was completed.
    'Hence, there is a more 'fundamental' reason for rejection!
End If
```

Now, this is still not all. "Task completed" is not equal to "task completed successfully". In the above example, we were scanning for NET1 network. Now, did we actually discover it? Find out by testing the value of the [wln.scanresultssid](#)⁵²⁶! For every task that may result in failure there is a way to know if the execution was successful or not.

```
'THE BULLETPROOF WAY...
...
While wln.task<>PL_WLN_TASK_IDLE 'waiting for the previous task to
complete...
Wend
If wln.scan("NET1")<>ACCEPTED Then
    'Handle this...
End If
While wln.task<>PL_WLN_TASK_IDLE 'waiting for the scan to complete...
Wend
'did we find this network?
If wln.scanresultssid<>"NET1" Then
    'Network was not found -- handle that...
End If
'start next task here
```

One problem with the code in the above examples is that it is, essentially, blocking. Your application is not doing anything useful while the Wi-Fi interface is scanning, then idling while the Wi-Fi is associating, and so on.

To take advantage of the event-driven nature of the system, you can base your Wi-Fi control on the [on_wln_task_complete](#)^[523] event which is generated each time a task is completed. Completed_task argument of the event handler carries the code of the event that has been completed. Therefore, you can advance through steps in this manner:

```
'THIS CODE TAKES FULL ADVANTAGE OF THE EVENT-DRIVEN NATURE OF THE SYSTEM
'-----
Sub On_sys_init
    ...
    wln.settxpower(14) 'issue the task and don't wait
End Sub

'-----
Sub On_wln_task_complete(completed_task As pl_wln_tasks)
    Select Case completed_task
        Case PL_WLN_TASK_SETTXPOWER:
            'here when wln.settxpower completes (we started it in the
on_sys_init)
            wln.scan("NET1") 'TX power set, now scan

        Case PL_WLN_TASK_SCAN:
            'scan completed, now associate

        wln.associate(wln.scanresultbssid,"NET1",wln.scanresultchannel,wln.scanres
ultbssmode)

        Case PL_WLN_TASK_ASSOCIATE:
            '... continue in this fashion

    End Select
End Sub

'-----
Sub On_wln_event(wln_event As pl_wln_events)
    'here we catch hardware problems and disassociations -- also
    asynchronously
End Sub
```

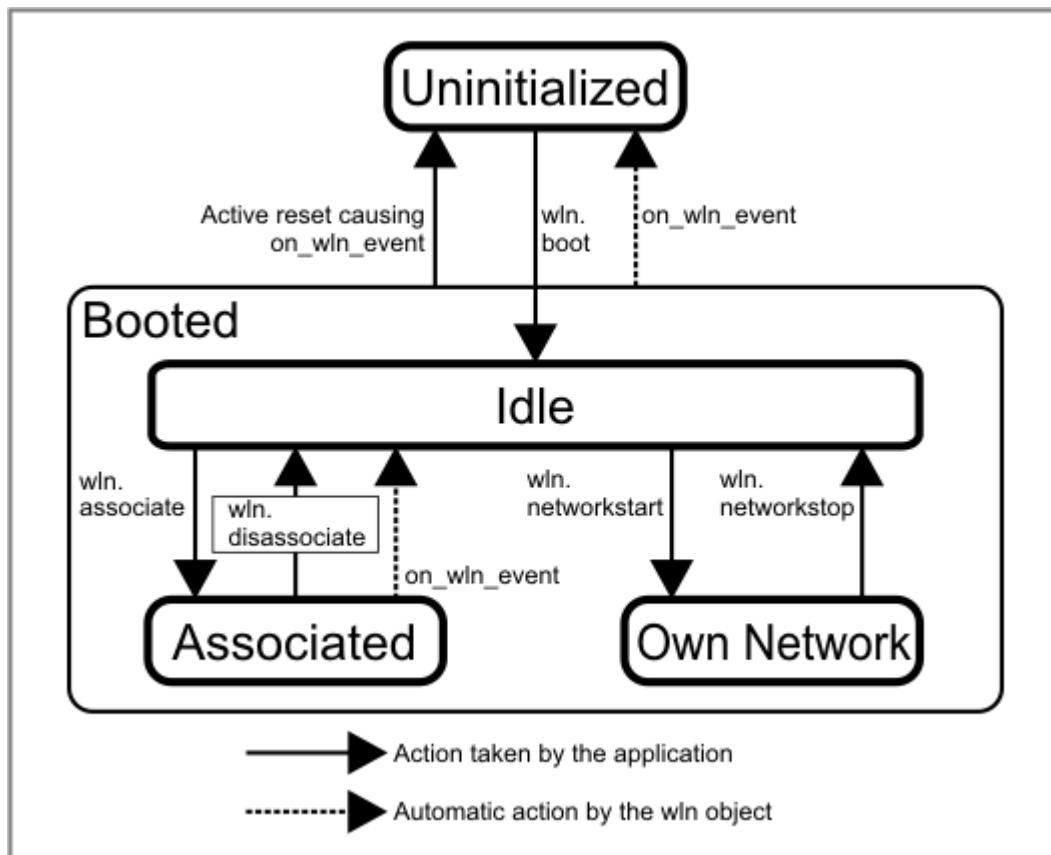
Notice the [on_wln_event](#)^[522] in the code above. It allows us to catch "problems".

Wln State Transitions

The Wi-Fi hardware may be in one of the following states:

- Uninitialized state ([wln.enabled](#)⁵¹⁹= 0- NO).
- Booted and idle (wln.enabled= 1- YES and [wln.associationstate](#)⁵¹⁴= 0- PL_WLN_NOT_ASSOCIATED).
- Booted and associated with a network, either infrastructure or ad-hoc (wln.enabled= 1- YES and wln.associationstate= 1- PL_WLN_ASSOCIATED).
- Booted and running an ad-hoc network of its own (wln.enabled= 1- YES and wln.associationstate= 2- PL_WLN_OWN_NETWORK).

The following diagram details possible state transitions.



The only way to advance from the uninitialized state into the booted state is through a successful boot. The process is described in [Bringing Up Wi-Fi Interface](#)⁵⁰⁴. The key method for the process is [wln.boot](#)⁵¹⁵.

There is no special method for powering down. Your application can only [reboot](#)⁵¹³ the Wi-Fi hardware, after which the boot process can be repeated. The [on_wln_event](#)⁵²² is generated when the GA1000 goes offline, either as a result of a deliberate reset, or in case the Wi-Fi hardware malfunctions.

Transition between the idle and associated states happens as a result of successful association. This is detailed in the [Associating With Selected Network](#)⁵¹¹ topic. The key method is [wln.associate](#)⁵¹³.

The [wln.disassociate](#)⁵¹⁸ method can be used to force disassociation. The wln object also detects the loss of association automatically, i.e. when the network in

question "disappears". In both cases, the [on_wln_event](#)^[522] event is generated.

The Wi-Fi interface can also [create its own ad-hoc network](#)^[512], which is achieved through the [wln.networkstart](#)^[521] method. [Terminating Own Ad-hoc Network](#)^[512] explains how to end this (in short, use [wln.networkstop](#)^[522]).

Notice that you cannot be associated and run your own network at the same time. These states are mutually exclusive.

Brining Up Wi-Fi Interface

A number of steps have to be taken in order to bring up the Wi-Fi interface. The following is a simplified code that demonstrates the process. Typically, the code would be in the [on_sys_init](#)^[220] event handler. We call the code simplified because it does not check for any error conditions.

The code includes the following steps:

- [Allocating buffer memory](#)^[505]
- [Applying reset](#)^[506]
- [Configuring Interface Lines](#)^[507]
- [Setting MAC address](#)^[507] (optional)
- [Selecting domain](#)^[508]
- [Booting up the hardware](#)^[508]
- [Setting IP, Gateway, and Netmask](#)^[509] (optional)
- [Setting TX power](#)^[509] (optional)

And now the code itself:

```
'BRINGING UP THE WI-FI MODULE (SIMPLIFIED)

Sub On_sys_init()

    '----- allocate buffers -----
    wln.buffrq(6)
    sys.buffalloc

    '----- issue hardware reset -----
    io.num=WLN_RST
    io.enabled=YES
    io.state=LOW
    io.state=HIGH

    '----- map interface lines -----
    wln.csmap=WLN_CS
    io.num=WLN_CS
    io.enabled=YES

    wln.dimap=WLN_DI

    wln.domap=WLN_Do
    io.num=WLN_Do
    io.enabled=YES

    wln.clkmap=WLN_CLK
    io.num=WLN_CLK
    io.enabled=YES

    '----- set MAC address (optional) -----
    wln.mac="0.100.110.120.130.140"

    '----- set domain -----
    wln.domain=PL_WLN_DOMAIN_FCC

    '----- boot up the GA1000 -----
    romfile.open("ga1000fw.bin")
    wln.boot(romfile.offset)

    '----- setup the IP, gateway, netmask -----
    wln.ip="192.168.1.86"
    wln.gatewayip="192.168.1.1"
    wln.netmask="255.255.255.0"

    '----- set TX power (optional) -----
    wln.settxpower(15)
    ...

End Sub
```

Allocating Buffer Memory

The first step is to allocate memory for a single buffer required by the `wln` object. This buffer is used to form outgoing packets and is necessary for correct operation. You never have to deal with this buffer directly -- it is handled internally by the `wln` object.

Buffer memory is allocated in pages. A page is 256 bytes of memory. Allocating memory for a buffer is a two-step process: First you have to request for a specific allocation (a number of pages) and then you have to perform the actual allocation. Request the size you need in pages using the [wln.buffrq](#)^[516] method.

The allocation method ([sys.buffalloc](#)^[217]) applies to all buffers previously specified, in one fell swoop:

```

Dim x As Byte
x = wln.buffrq(6) ' request 6 pages for the wln buffer. Out will then
contain how many can actually be allocated.
' .... Allocation requests for buffers of other objects ....
sys.buffalloc 'perform actual memory allocation, as per previous requests.

```

Actual memory allocation takes up to 100ms, so it is usually done just once, on boot, for all required buffers.

You may not always get the full amount of memory you have requested. Memory is not an infinite resource, and if you have already requested (and received) allocations for 95% of the memory for your platform, your next request will get up to 5% of memory, even if you requested for 10%.

Current wln buffer size in bytes can always be checked with the [wln.buffsize](#)^[516] read-only property.

Note that wln buffer size can't be changed when the Wi-Fi hardware is already [booted](#)^[508].

How many pages should the wln buffer get?

The size of the wln buffer directly dictates the maximum size of network packets that the wln object will be able to send (this buffer has nothing to do with incoming packets). Up to 100 bytes of the buffer space are required for various packet headers, and the rest is available to packet payload. For example, if you have allocated 2 pages for the buffer, then the buffer size is 512 bytes. Hence, maximum payload size cannot exceed 412 bytes.

For TCP communications, the size of individual packets is not that critical. The beauty of TCP is that it can work with whatever buffer space is available. It is true that the bigger the buffer, the better TCP throughput is. In reality, you will stop feeling any improvement in TCP performance once your wln buffer size exceeds 2 or 3 pages.

The UDP is another matter entirely. If you want to be able to send the packets of a certain size, then you must make sure that you have created an adequate wln buffer. For example, the DHCP protocol is based on UDP packets. The size of UDP-DHCP packets can be as large as 1400 bytes. If you are planning to use DHCP on the wln interface, you will have to allocate the wln buffer that allows the UDP packets with the payload of at least 1400 bytes. Therefore, you will need at least 1500 bytes of buffer space. Round this to 256-byte pages, and we arrive at required buffer size of 6 pages.

Applying Reset

The Wi-Fi module requires a hardware reset for correct operation and there is a dedicated reset (RST) interface pin for that. Any I/O line of your Tibbo module can be used to control the reset line.

To reset the Wi-Fi module, enable the GPIO line that controls the reset (let's assume it is GPIO #51) and set it to LOW first, HIGH next. The reset is complete!

```

'reset the Wi-Fi module
io.num=PI_IO_NUM_51
io.enabled=YES      'enable this line
io.state=LOW        'apply reset
io.state=HIGH       'remove reset

```

Configuring Interface Lines

The Wi-Fi module interacts with your BASIC-programmable module through an SPI interface. The SPI interface has four signals: chip select (CS), clock (CLK), data in (DI), and data out (DO). Any four I/O lines of your programmable module can be selected to control the SPI interface. Choosing I/O lines for the job is called "mapping". The CS, CLK, and DO lines must be configured as outputs, and the DI line will be an input:

```
'configure the CS line
wln.csmap=WLN_CS
io.num=WLN_CS
io.enabled=YES

'configure the DI line
wln.dimap=WLN_DI

'configure the DO line
wln.domap=WLN_DO
io.num=WLN_DO
io.enabled=YES

'configure the CLK line
wln.clkmap=WLN_CLK
io.num=WLN_CLK
io.enabled=YES
```

Note that mapping can't be changed when the Wi-Fi hardware is already [booted](#)⁵⁰⁸ (i.e. [wln.enabled](#)⁵¹⁹= 1- YES).

Setting MAC Address (Optional)

Every network interface needs its own MAC, and the Wi-Fi port is no exception. The [wln.mac](#)⁵²⁰ property exists for this purpose. Your Wi-Fi module already carries a MAC address onboard -- it is preset during manufacturing, so you don't actually have to take care of the MAC. Leave the wln.mac at its default pre-boot value of "0.0.0.0.0.0", [boot up](#)⁵⁰⁸ the Wi-Fi interface ([wln.boot](#)⁵¹⁵), and the wln.mac will be updated with the pre-assigned address that is stored inside the Wi-Fi module.

You can use another MAC if you want, too. Set the desired MAC address before booting up the Wi-Fi, and this MAC will be used instead of the pre-assigned one. That is, if wln.boot is called after you set the wln.mac to anything but "0.0.0.0.0.0", then your MAC will be used instead:

```
'set the mac address
wln.mac="0.1.2.3.100.200" 'override pre-assigned MAC with another address
...
romfile.open("ga1000fw.bin")
wln.boot(romfile.offset) 'the hardware will start using your MAC
```

The pre-assigned MAC inside the Wi-Fi module will not be altered. It is always there and can be called up by leaving the wln.mac at "all zeroes", then booting the Wi-Fi hardware.

A bit of info on MACs

The MAC address can be either "globally unique" or "locally administered". There is also a provision for "unicast" and "multicast" addressing. You can find more information on this here: http://en.wikipedia.org/wiki/Mac_address.

Your organization can purchase a block of globally unique addresses, or choose to assign random locally administered addresses. In the latter case, set the most significant byte of the address to 2, and choose random values for the remaining 5 bytes ([random](#)^[21] function will help). Note that each device you are using should have a unique MAC address. It is a good idea to generate the MAC once, and store it in the EEPROM memory (see the [stor](#)^[38] object). The MAC can then be retrieved on each boot and written into the [wln.mac](#)^[520] property.

Note that the MAC address of the Wi-Fi interface can't be set when the hardware is already booted (wln.enabled= 1- YES).

Selecting Domain

Wireless communications and channels are tightly regulated in every country on Earth, and this applies to Wi-Fi networks as well. Not every one of 14 pre-defined Wi-Fi frequencies is allowed to be used in every country. It is your responsibility to set a correct "domain" for your Wi-Fi device. This is done through the [wln.domain](#)^[518] property. Supported domains are FCC, EU, JAPAN, and "OTHER".

Booting Up the Hardware

Booting up the Wi-Fi hardware is done through the [wln.boot](#)^[515] method. The Wi-Fi hardware does not have a ROM or flash memory and its internal processor executes its firmware from RAM. Before the Wi-Fi module can start working, you need to upload this firmware into the Wi-Fi module, and this is what wln.boot really does.

The firmware file is called "ga1000fw.bin" (the file can be downloaded from Tibbo website). The file must be added to your Tibbo Basic project as a binary [resource file](#)^[20]. (see also [Add File to Project](#)^[128]).

Access to resource files is through the [romfile](#)^[375] object. First, you open the "ga1000fw.bin" file with the [romfile.open](#)^[379] method, then pass the pointer to this file (value of the [romfile.offset](#)^[376] R/O property) to the wln.boot method:

```
'boot it up
romfile.open("ga1000fw.bin")
If wln.boot(romfile.offset)=NG Then
    'something is wrong, react to this
    ...
End If
```

The boot takes less than 1 second to complete. The method will return 0- OK if the boot was completed successfully, or 1- NG if the boot failed. The boot may fail for several reasons. The Wi-Fi hardware may not be powered, connected improperly, mapped incorrectly, or malfunction. Additionally, the boot will fail if the Wi-Fi hardware is already booted.

Setting IP, Gateway, and Netmask (Optional)

The Wi-Fi is a separate network interface and so it has its own IP address, which is set using the [wln.ip](#)^[520] property. This address is different from the IP address of the Ethernet interface (see [net.ip](#)^[270]). We noticed that many people find it "unusual" that Tibbo hardware device would turn out to have two IP addresses. In fact, this is completely normal. On the PC, every network interface has an IP of its own as well.

Technically speaking, IP address configuration can be done at any time. This topic has been placed into the [Bringing Up Wi-Fi Interface](#)^[504] section to remind you that the IP of the wln object has to be set, if not right after the [boot](#)^[508], then at a later point. If your application uses a static IP, then setting it in the boot section of your code is a good idea. If the application obtains the IP address through DHCP, then the IP can only be set after communicating with the DHCP server, and this will only be possible after successful [association](#)^[511]. You may even need to set the IP address repeatedly if your product switches between different networks (access points).

There are also [wln.gatewayip](#)^[520] and [wln.netmask](#)^[511] that may need to be set along with the IP address. This is optional and is only required if your device will have to establish outgoing connections to the network hosts outside of your LAN.

Note that the IP, gateway IP, and netmask of the Wi-Fi interface can't be set when there is at least one open socket in your system ([sock.statesimple](#)^[358]<> 0-PL_SSTS_CLOSED for any socket).

Setting TX Power (Optional)

The output power of the Wi-Fi hardware can be adjusted in 12 steps. The [wln.settxpower](#)^[527] method is provided for that purpose. The power value roughly corresponds to dB. The lowest output power is set with `wln.settxpower(4)` and the highest power is set with `wln.settxpower(15)`. Lower power reduces the current consumption of the Wi-Fi module, but not by much. We recommend that you just use the default power of 15.

Note that setting TX power is a [wln task](#)^[500] and there is a certain correct way of handling tasks.

Scanning for Wi-Fi Networks

Scanning allows you to discover all networks in your device's "neighborhood" and also learn about their operating parameters, such as the name, RF channel, etc. The [wln.scan](#)^[524] method is provided for this purpose. Note that scanning is a [wln task](#)^[500] and there is a certain correct way of handling tasks.

The `wln.scan` accepts a single argument of string type. Presence or absence of this argument defines the "operating mode" of the method.

When called with an empty string, the `wln.task` will attempt to find all available wireless networks. After the task completes, a [wln.scanresultssid](#)^[526] property will contain a comma-separated list of network names:

```
'scan for available network
Dim s As String
...
wln.scan("")
While wln.task<>PL_WLN_TASK_IDLE
Wend
s=wln.scanresultssid 'the list of networks will be copied into s
```

After the execution of the above, s string may contain something like this: "TIBBO, c1100_1,WNET2".

When called with the argument set to the name of a particular network, the wln.scan method will return additional data on this network:

- [Wln.scanresultssid](#)^[526] R/O property will contain the name of the specified network, or nothing if the network wasn't found. If the network was found, the following four properties will have additional data on this network:
- [Wln.scanresultbssid](#)^[525] R/O property will contain the BSSID of the specified network.
- [Wln.scanresultbssmode](#)^[525] R/O property will contain the BSS mode of the network.
- [Wln.scanresultchannel](#)^[526] R/O property will return the number of the RF channel on which the network operates.
- [Wln.scanresultrssi](#)^[526] R/O property will contain the strength of the RF signal received from the specified network.

The scanning cannot be performed while the Wi-Fi interface is in the associated state ([wln.associationstate=](#)^[514] 1- PL_WLN_ASSOCIATED) or is running its own ad-hoc network (wln.associationstate= 2- PL_WLN_OWN_NETWORK).



Don't know what SSID in wln.scanresult means? How about BSSID or RSSI? The [Wi-Fi Parlance Primer](#)^[500] will tell you what it all means!

Setting WEP Mode and Key

If the network you are [associating](#)^[511] with (creating) uses (is supposed to be using) [WEP](#)^[500], then you need to set the WEP prior to associating with (creating) the network. The [wln.setwep](#)^[527] method allows you to specify the key and the WEP mode. Note that wln.setwep is a [wln task](#)^[500] and there is a certain correct way of handling tasks.

The mode can be either OFF, 64-bit WEP, or 128-bit WEP. WEP key is entered as a HEX string, not ASCII string. Each character in a string represents one HEX digit: 0..9 or A..F (a..f). The key has a fixed length: 10 HEX digits for WEP-64 or 26 HEX digits for WEP-128. If your key is too short, it will be padded with zeroes, if the key is too large it will be truncated.

Here is the code example that sets the Wi-Fi to WEP-128 mode:

```
'set WEP-128
wln.setwep("11111111111111111111111111",PL_WLN_WEP_MODE_128) 'we love to
choose difficult keys
While wln.task<>PL_WLN_TASK_IDLE
Wend
```

Note that the WEP mode and key can't be changed while the Wi-Fi interface is in the associated state ([wln.associationstate=](#)^[514] 1- PL_WLN_ASSOCIATED) or is running its own network (wln.associationstate= 2- PL_WLN_OWN_NETWORK).



Wi-Fi devices routinely define four WEP keys, but Tibbo hardware only uses a single key (key 1).

Associating With Selected Network

Association is a process by which your Wi-Fi device establishes a network link with an access point or another wireless station running an ad-hoc network.

The association process is initiated using the [wln.associate](#)^[513] method. Association is a required step before you will be able to send and receive the data over the Wi-Fi. Note that wln.associate is a [wln task](#)^[500] and there is a certain correct way of handling tasks. Also, you can only associate from an idle state ([wln.associationstate](#)^[514]= 0- PL_WLN_NOT_ASSOCIATED).

Prior to associating, you need to set the [WEP](#)^[509] (if required). You also need to know several key parameters about the network you are associating with:

- SSID (name) of this network;
- BSSID ("MAC") of this network;
- Channel on which this network operates;
- BSS mode of this network (whether the network is infrastructure or ad-hoc).

Normally, the SSID is known, but BSSID, channel, and BSS mode require some digging. The easiest way to sniff out this info is through [scanning](#)^[509]. The [wln.scan](#)^[524] fills out [wln.scanresultbssid](#)^[525], [wln.scanresultchannel](#)^[526], and [wln.scanresultbssmode](#)^[525] R/O properties. You only need to put them to good use, as shown here:

```
'connect to the access point named TIBBO.
wln.wepmode=PL_WLN_WEP_MODE_DISABLED 'This one has WEP switched off --
don't even count on it in real life!

wln.scan("TIBBO") 'scanning for a specific network will give us necessary
parameters
While wln.task<>PL_WLN_TASK_IDLE
Wend
If wln.scanresultssid<>"" Then
    'scanning failed for some reason
End If

'now can associate: 'wln.scanresult' properties contain necessary data
after the scanning
wln.associate(wln.scanresultbssid, wln.scanresultssid,
wln.scanresultchannel, wln.scanresultbssmode)
While wln.task<>PL_WLN_TASK_IDLE
Wend
If wln.associationstate=PL_WLN_ASSOCIATED Then
    'successful association!
...
End If
```

After the association task is completed you have to check the association result. Mere task completion does not indicate success! The [wln.associationstate](#)^[514] will provide the indication.

Once you have achieved association, you can [communicate](#)^[512] over the Wi-Fi interface. Re-association is not allowed, you need to [disassociate](#)^[512] first. Disassociation can also happen automatically (for example, if the access point goes offline or out of range), and the wln object will be able to [detect](#)^[513] this.

When the wln is in the associated state, the [wln.rssi](#)^[524] read-only property is constantly updated with the strength of the signal being received from the currently used wireless network. Do not confuse this with the [wln.scanresultrss](#)^[526] property which returns the signal strength of a particular network obtained in the [scan process](#)^[509].

Creating Own Ad-hoc Network

Rather than [associating](#)^[511] with somebody else's network, the Wi-Fi interface can create an ad-hoc network of its own and have other devices associate with you. This is done by using [wln.networkstart](#)^[521] method. Note that wln.networkstart is a [wln task](#)^[500] and there is a certain correct way of handling tasks. Also, you can only create your own network from an idle state ([wln.associationstate](#)^[514]= 0-PL_WLN_NOT_ASSOCIATED).

To start a network, you only need to make up your mind regarding its name and operating channel:

```
'take control and run our own network
wln.networkstart("VOICEOFFREBELS", 6)
While wln.task<>PL_WLN_TASK_IDLE
Wend
```

Communicating via WIn Interface

Actual data exchange over the Wi-Fi link falls outside the responsibilities of the wln object. This is the task of the [sock](#)^[274] object. This object has a set of properties that define whether a particular socket will be listening on the Wi-Fi interface ([sock.allowedinterfaces](#)^[327] property), establish an outgoing connection through the Wi-Fi interface ([sock.targetinterface](#)^[359] property). See also: [sock.currentinterface](#)^[331] read-only property.

Disassociating From the Network

To disassociate from the network, use the [wln.disassociate](#)^[518] method. Note that wln.disassociate is a [wln task](#)^[500] and there is a certain correct way of handling tasks.

```
'disassociate now
wln.disassociate
While wln.task<>PL_WLN_TASK_IDLE
Wend
```

Disassociation can happen automatically, for example, when the access point goes offline or out of range. This will be [detected](#)^[513] by the wln object.

Terminating Own Ad-hoc Network

Use [wln.networkstop](#)^[522] to terminate your ad-hoc network. We are talking about the network your hardware has created -- you can't terminate an ad-hoc network of someone else. Note that wln.networkstop is a [wln task](#)^[500] and there is a certain correct way of handling tasks.

Rebooting

Rebooting is done by applying a [hardware reset](#)^[506] to the Wi-Fi module. When the Wi-Fi interface add-on goes offline, the `wln` object will [detect](#)^[513] this in a matter of milliseconds. Your application can stay and wait for this:

```
'reset the Wi-Fi module (we assume that the RST line is connected to I/O
line #51)
io.num=PL_IO_NUM_51
io.enabled=YES      'enable this line
io.state=LOW        'apply reset
io.state=HIGH       'remove reset

'wait for the wln object to detect this
While wln.enabled=YES
Wend

'OK, now can repeat the boot process
```

Detecting Disassociation or Offline State

The `wln` object automatically detects disassociation from the wireless network and powering-off of the Wi-Fi hardware. [On_wln_event](#)^[522] event is fired up if either condition is detected. In response to this event, your application can re-initialize the Wi-Fi hardware and/or re-associate with the wireless network.

Properties, Methods, Events

Properties, methods, and events of the `wln` object.

.Associate Method

Function: Causes the Wi-Fi interface to attempt association with the specified wireless network.

Syntax: `wln.associate(byref bssid as string, byref ssid as string, channel as byte, bssmode as pl_wln_bss_modes) as accepted_rejected`

Returns: One of accepted_rejected constants:

0- ACCEPTED.

1- REJECTED.

See Also: [Associating With Selected Network](#)^[511], [Setting WEP Mode and Key](#)^[510], [Wln Tasks](#)^[500]

Part	Description
bssid	The BSSID ("MAC address") of the network with which to associate.
ssid	The name of the target network with which to associate.
chan	Channel on which the target network is operating.
nel	

ssid The name of the target network with which to associate.

chan Channel on which the target network is operating.

bssm Network mode:
ode 0- PL_WLN_BSS_MODE_INFRASTRUCTURE: This is an infrastructure network (access point).
1- PL_WLN_BSS_MODE_ADHOC: This is an ad-hoc (device-to-device) network.

Details

Prior to performing an association attempt, the application must preset the WEP mode and key if required (see [wln.setwep](#)). For successful association, all method arguments must be specified correctly. The ssid is the name of the target wireless network. Remaining parameters can be obtained through the [wln.scan](#) method: when the method is performed with the network name specified, it stores the bssid, channel, and bssmode of this network in [wln.scanresultbssid](#), [wln.scanresultchannel](#), and [wln.scanresultbssmode](#) R/O properties. For more information see [Associating With Selected Network](#).

The association process is a [task](#). As such, the wln.associate will be rejected (return 1- REJECTED) if another task is currently in progress. The task will also be rejected if the Wi-Fi interface is already in the associated state ([wln.associationstate](#)= 1- PL_WLN_ASSOCIATED), is running its own network ([wln.associationstate](#)= 2- PL_WLN_OWN_NETWORK), or if the Wi-Fi hardware is not online ([wln.enabled](#)= 0- NO). The method will return 0- ACCEPTED if the task is accepted for processing.

The task is completed when the [wln.task](#) R/O property becomes 0- PL_WLN_TASK_IDLE. The [on_wln_task_complete](#) event will also be generated at that time.

Task completion does not imply success -- association result has to be verified by reading the state of the wln.associationstate read-only property after the task is completed.

.Associationstate R/O Property

Function:	Indicates whether the Wi-Fi interface is idle, associated with another network, or running its own ad-hoc network.
Type:	Enum (pl_wln_association_states, byte)
Value Range:	0- PL_WLN_NOT_ASSOCIATED (default): The Wi-Fi interface is not associated with any wireless network and is not running its own ad-hoc network. 1- PL_WLN_ASSOCIATED: The Wi-Fi interface is associated with a wireless network. 2- PL_WLN_OWN_NETWORK: The Wi-Fi interface is running its own ad-hoc network.
See Also:	Associating With Selected Network , Disassociating From the Network , Creating Own Ad-hoc Network , Terminating Own Ad-hoc Network , Detecting Disassociation or Power-down

Details

After the successful association, which is initiated through the [wln.associate](#)^[513] method, the value of this property changes to 1- PL_WLN_ASSOCIATED. The value is reset back to 0- PL_WLN_NOT_ASSOCIATED if disassociation occurs. Every time this happens, an [on_wln_event](#)^[522] event is generated with its wln_event argument set to 1- PL_WLN_EVENT_DISASSOCIATED. Disassociation may happen for a number of reasons: it can be induced through the [wln.disassociate](#)^[518] method or forced by the access point. The disassociation will also happen if the access point goes offline or out of range, or if the Wi-Fi hardware is powered down, disconnected, or malfunctions.

After the Wi-Fi interface succeeds in creating its own ad-hoc network (see [wln.networkstart](#)^[521]), the value of this property becomes 2- PL_WLN_OWN_NETWORK. The value is reset back to 0- PL_WLN_NOT_ASSOCIATED when the ad-hoc network is terminated ([wln.networkstop](#)^[522]).

.Boot Method

Function:	Boots up the Wi-Fi interface, which involves sending to the Wi-Fi hardware a firmware file for its embedded processor.
Syntax:	wln.boot(offset as dword) as ok_ng
Returns:	One of ok_ng constants: 0- OK: completed successfully. 1- NG: boot failed.
See Also:	Booting Up the Hardware ^[508]

Part Description

offset	Offset of the "ga1000fw.bin" file within the compiled binary of your project. The offset is obtained using the romfile.offset ^[378] read-only property.
--------	--

Details

The "ga1000fw.bin" file must be present in your project as a binary resource file. To obtain correct offset value, open the file first: [romfile.open](#)^[379]("wln_fwar.bin"). After that, do `wln.boot(romfile.offset)`.

The boot process is very fast and will be completed in less than 1 second.

This method will return 0- OK when the boot completes successfully. At the same time, the [wln.enabled](#)^[519] will become 1- YES. The method will return 1- NG if the boot fails, in which case the `wln.enabled` will remain at 0- NO. This will happen if the Wi-Fi hardware is not connected properly, [mapped](#)^[507] incorrectly, not powered, malfunctions, or is operational already.

.Bssmode Property

Obsolete.

This parameter is now supplied directly, as the argument for the [wln.associate](#)^[513] method.

.Buffrq Method

Function:	Pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the TX buffer of the wln object.
Syntax:	wln.buffrq(numpages as byte) as byte
Returns:	Actual number of pages that can be allocated (byte).
See Also:	Allocating Buffer Memory 

Part Description

num	Requested numbers of buffer pages to allocate.
page	
s	

Details

Actual allocation happens when the [sys.buffalloc](#)²¹⁷ method is used. The wln object will be unable to operate properly if its TX buffer has inadequate capacity (more on this in the [Allocating Buffer Memory](#)⁵⁰⁵ topic). The Wi-Fi interface will only be able to successfully send out packets that fit in this buffer; larger packets will be truncated. There is no need to allocate any buffer for packet reception.

Buffer allocation will not work if the Wi-Fi hardware is already operational ([wln.enabled](#)⁵¹⁹= 1- YES). Executing sys.buffalloc at this time will leave the buffer size unchanged. Therefore, buffer allocation must happen before the Wi-Fi hardware is booted up with the [wln.boot](#)⁵¹⁵ method.

Actual current buffer size can be verified through the [wln.buffsize](#)⁵¹⁶ read-only property.

.Buffsize R/O Property

Function:	Returns current capacity (in bytes) of the wln object's TX buffer.
Type:	Word
Value Range:	0-65535, default = 0 (0 bytes).
See Also:	Allocating Buffer Memory 

Details

Buffer capacity can be changed through the [wln.buffrq](#)⁵¹⁶ method followed by the [sys.buffalloc](#)²¹⁷ method.

The Wi-Fi interface will only be able to successfully send out packets that fit in the TX buffer; larger packets will be truncated. There is no need to allocate any buffer for packet reception.

.Clkmap Property

Function:	Sets/returns the number of the I/O line which will control the clock (CLOCK) input of the Wi-Fi module's SPI interface.
Type:	Enum (pl_io_num, byte)
Value Range:	Platform-specific. See the list of pl_io_num constants in the platform specifications. Default = PL_IO_NULL (NULL line).
See Also:	Configuring Interface Lines 

Details

This selection cannot be changed once the Wi-Fi hardware is already operational ([wln.enabled](#)  = 1- YES). Correct mapping must be specified before attempting to boot the Wi-Fi hardware, or [wln.boot](#)  will fail.

.Csmap Property

Function:	Sets/returns the number of the I/O line which will control the chip select (CS) input of the Wi-Fi module's SPI interface.
Type:	Enum (pl_io_num, byte)
Value Range:	Platform-specific. See the list of pl_io_num constants in the platform specifications. Default = PL_IO_NULL (NULL line).
See Also:	Configuring Interface Lines 

Details

This selection cannot be changed once the Wi-Fi hardware is already operational ([wln.enabled](#)  = 1- YES). Correct mapping must be specified before attempting to boot the Wi-Fi hardware, or [wln.boot](#)  will fail.

.Defaultibsschannel Property

Obsolete.

This parameter is now supplied directly, as the argument for the [wln.networkstart](#)  method.

.Dimap Property

Function:	Sets/returns the number of the I/O line which will control the data in (DI) input of the Wi-Fi module's SPI interface.
Type:	Enum (pl_io_num, byte)
Value Range:	Platform-specific. See the list of pl_io_num constants in the platform specifications. Default = PL_IO_NULL (NULL line).

See Also: [Configuring Interface Lines](#)

Details

This selection cannot be changed once the Wi-Fi hardware is already operational ([wln.enabled](#)= 1- YES). Correct mapping must be specified before attempting to boot the Wi-Fi hardware, or [wln.boot](#) will fail.

.Disassociate Method

Function: Causes the Wi-Fi interface to commence disassociation from the wireless network.

Syntax: **wln.disassociate() as accepted_rejected**

Returns: One of accepted_rejected constants:

0- ACCEPTED.

1- REJECTED.

See Also: [Disassociating From the Network](#), [Wln Tasks](#)

Details

The disassociation process is a [task](#). As such, the wln.disassociate will be rejected (return 1- REJECTED) if another task is currently in progress. The task will also be rejected if the Wi-Fi hardware is not online ([wln.enabled](#)= 0- NO). The method will return 0- ACCEPTED if the task is accepted for processing.

The task is completed when the [wln.task](#) R/O property becomes 0- PL_WLN_TASK_IDLE. The [on wln task complete](#) event will also be generated at that time. Additionally, the [on wln event](#) event will be generated with PL_WLN_EVENT_DISASSOCIATED argument.

.Domain Property

Function: Selects the domain (area of the world) in which this device is operating. This defines the list of channels on which the Wi-Fi interface will be allowed to associate with wireless networks.

Type: Enum (pl_wln_domains, byte)

Value Range: 0- PL_WLN_DOMAIN_FCC (**default**): FCC domain (US, Canada, Taiwan...). Allowed channels: 1-11.

1- PL_WLN_DOMAIN_EU: European Union. Allowed channels: 1-13.

2- PL_WLN_DOMAIN_JAPAN: Japan. Allowed channels: 1-14.

3- PL_WLN_DOMAIN_OTHER: All other countries. Allowed channels: 1-14.

See Also: [Selecting Domain](#), [Scanning for Wi-Fi Networks](#), [Associating With Selected Network](#), [Creating Own Ad-hoc Network](#)

Details

This property can't be changed while the Wi-Fi hardware is operational ([wln.enabled](#)⁵¹⁸= 1- YES). Note that domain selection only affects association ([wln.associate](#)⁵¹³), not the scanning process ([wln.scan](#)⁵²⁴) or the ability of the Wi-Fi interface to start its own ad-hoc network ([wln.networkstart](#)⁵²¹) on whatever channel you specify.

.Domap Property

Function:	Sets/returns the number of the I/O line which will control the data out (DO) output of the Wi-Fi module's SPI interface.
Type:	Enum (pl_io_num, byte)
Value Range:	Platform-specific. See the list of pl_io_num constants in the platform specifications. Default = PL_IO_NULL (NULL line).
See Also:	Configuring Interface Lines ⁵⁰⁷

Details

This selection cannot be changed once the Wi-Fi hardware is already operational ([wln.enabled](#)⁵¹⁸= 1- YES). Correct mapping must be specified before attempting to boot the Wi-Fi hardware, or [wln.boot](#)⁵¹⁵ will fail.

.Enabled R/O Property

Function:	Indicates whether the Wi-Fi interface is operational.
Type:	Enum (no_yes, byte)
Value Range:	0- NO (default): The Wi-Fi interface is not operational. 1- YES: The Wi-Fi interface is operational.
See Also:	Booting Up the Hardware ⁵⁰⁸ , Rebooting ⁵¹³ , Detecting Disassociation or Offline State ⁵¹³

Details

The Wi-Fi hardware becomes operational after a successful boot using the [wln.boot](#)⁵¹⁵ method, at which time the wln.enabled is set to 0- NO. The Wi-Fi interface is disabled and the wln.enabled is reset to 0- NO if the Wi-Fi hardware is disconnected, powered down, or if it malfunctions. When this happens, the [on_wln_event](#)⁵²² event is generated with its wln_event argument set to 0- PL_WLN_EVENT_DISABLED.

.Gatewayip Property

Function:	Sets/returns the IP address of the default gateway for the Wi-Fi interface of your device.
Type:	Dot-decimal string
Value Range:	Any IP address, such as "192.168.1.1". Default = "0.0.0.0".
See Also:	Setting IP, Gateway, and Netmask ^[509] , wln.ip ^[520] , wln.netmask ^[521]

Details

This property can only be written to when no socket is engaged in communications through the Wi-Fi interface, i.e. there is no socket for which [sock.statesimple](#)^[358]<> 0- PL_SSTS_CLOSED and [sock.currentinterface](#)^[33]= 2- PL_INTERFACE_WLN.

.Ip Property

Function:	Sets/returns the IP address of the Wi-Fi interface of your device.
Type:	Dot-decimal string
Value Range:	Any valid IP address, such as "192.168.100.40". Default = "1.0.0.1".
See Also:	Setting IP, Gateway, and Netmask ^[509] , wln.gatewayip ^[520] , wln.netmask ^[521]

Details

This property can only be written to when no socket is engaged in communications through the Wi-Fi interface, i.e. there is no socket for which [sock.statesimple](#)^[358]<> 0- PL_SSTS_CLOSED and [sock.currentinterface](#)^[33]= 2- PL_INTERFACE_WLN.

.Mac Property

Function:	Sets/returns the MAC address of the Wi-Fi interface.
Type:	Dot-decimal string
Value Range:	Any valid MAC address, i.e. "0.1.2.3.4.5". Default = "0.0.0.0.0.0".
See Also:	Setting MAC Address ^[507]

Details

There are certain rules on MAC address selection -- see [Setting MAC Address](#)^[520] topic for details.

This property can only be written to while the Wi-Fi hardware is not operational ([wln.enabled](#)^[519]= 0- NO) and when no socket is engaged in communications

through the Wi-Fi interface, i.e. there is no socket for which `sock.statesimple`^[358]<> 0- PL_SSTS_CLOSED and `sock.currentinterface`^[33]= 2- PL_INTERFACE_WLN.

The GA1000 add-on board already has a proper MAC address internally. To use this MAC, leave the wln.mac at "0.0.0.0.0" when booting up the wln interface with the `wln.boot`^[515] method. After a successful boot, the wln.mac property will contain the MAC address obtained from the GA1000. Alternatively, set your own MAC address before calling the wln.boot. This new address will be used, as long as it is not "0.0.0.0.0".

.Netmask Property

Function:	Sets/returns the netmask for the Wi-Fi interface of your device.
Type:	Dot-decimal string
Value Range:	Any valid netmask, such as "255.255.255.0". Default= "0.0.0.0".
See Also:	Setting IP, Gateway, and Netmask ^[509] , wln.ip ^[520] , wln.gatewayip ^[520]

Details

This property can only be written to when no socket is engaged in communications through the Wi-Fi interface, i.e. there is no socket for which `sock.statesimple`^[358]<> 0- PL_SSTS_CLOSED and `sock.currentinterface`^[33]= 2- PL_INTERFACE_WLN.

.Networkstart Method

Function:	Causes the Wi-Fi interface to attempt starting its own ad-hoc network.
Syntax:	<code>wln.networkstart(byref ssid as string, channel as byte) as accepted_rejected</code>
Returns:	One of accepted_rejected constants: 0- ACCEPTED. 1- REJECTED.
See Also:	Creating Own Ad-hoc network ^[512] , Setting WEP Mode and Key ^[510] , Wln Tasks ^[500]

Part Description

ssid	The name of the ad-hoc network to create.
chan nel	Channel on which the new ad-hoc network will operate.

Details

Prior to creating an ad-hoc network, the application must preset the WEP mode and key if required (see [wln.setwep](#)^[527]).

Ad-hoc network creation is a [task](#). As such, the `wln.networkstart` will be rejected (return 1- REJECTED) if another task is currently in progress. The task will also be rejected if the Wi-Fi interface is in the associated state (`wln.associationstate`=1- PL_WLN_ASSOCIATED), already runs its own network (`wln.associationstate`=2- PL_WLN_OWN_NETWORK), or if the Wi-Fi hardware is not online (`wln.enabled`=0- NO). The method will return 0- ACCEPTED if the task is accepted for processing.

The task is completed when the `wln.task` R/O property becomes 0-PL_WLN_TASK_IDLE. The [on_wln_task_complete](#) event will also be generated at that time.

Task completion does not imply success -- the result has to be verified by reading the state of the `wln.associationstate` read-only property after the task is completed.

.Networkstop Method

Function:	Causes the Wi-Fi interface to commence the termination of its own ad-hoc network.
Syntax:	wln.networkstop() as accepted_rejected
Returns:	One of accepted_rejected constants: 0- ACCEPTED. 1- REJECTED.
See Also:	Terminating Own Ad-hoc Network , Wln Tasks

Details

Ad-hoc network termination process is a [task](#). As such, the `wln.networkstop` will be rejected (return 1- REJECTED) if another task is currently in progress. The task will also be rejected if the Wi-Fi hardware is not online (`wln.enabled`=0- NO). The method will return 0- ACCEPTED if the task is accepted for processing.

The task is completed when the `wln.task` R/O property becomes 0-PL_WLN_TASK_IDLE. The [on_wln_task_complete](#) event will also be generated at that time.

On_wln_event Event

Function:	Generated when the <code>wln</code> object detects disassociation from the wireless network or the Wi-Fi hardware is disconnected, powered-down, or is malfunctioning.
Declaration:	on_wln_event(wln_event as pl_wln_events)
See Also:	Detecting Disassociation or Offline State

Part	Description
-------------	--------------------

wln_event Registered event:
 0- PL_WLN_EVENT_DISABLED: Wi-Fi hardware has been disconnected, powered down, or is malfunctioning.
 1- PL_WLN_EVENT_DISASSOCIATED: Wi-Fi interface has been disassociated from the wireless network.

Details

Multiple on_wln_event events may be waiting in the event queue.

On_wln_task_complete Event

Function: Generated when the Wi-Fi interface completes executing a given task.

Declaration: **on_wln_task_complete(completed_task as pl_wln_tasks)**

See Also: [Wln Tasks](#)

Part	Description
completed_task	The task just completed: 1- PL_WLN_TASK_SCAN: Scan task completed (this task is initiated by the wln.scan method). 2- PL_WLN_TASK_ASSOCIATE: Association task completed (this task is initiated by the wln.associate method). 3- PL_WLN_TASK_SETTXPOWER: TX power adjustment task completed (this task is initiated by the wln.settxpower method). 4- PL_WLN_TASK_SETWEP: WEP mode and keys setup task completed (this task is initiated by the wln.setwep method). 5- PL_WLN_TASK_DISASSOCIATE: Disassociation task completed (this task is initiated by the wln.disassociate method). 6- PL_WLN_TASK_NETWORK_START: Ad-hoc network creation completed (this task is initiated by the wln.networkstart method). 7- PL_WLN_TASK_NETWORK_STOP: Ad-hoc network termination completed (this task is initiated by the wln.networkstop method).

Details

The [wln.task](#) read-only property changes to 0- PL_WLN_TASK_IDLE along with this event generation. The wln object will only accept another task for execution after the previous task has been completed.

Multiple on_wln_task_complete events may be waiting in the event queue.

.Rssi R/O Property

Function:	Indicates the strength of the signal being received from the wireless network that the Wi-Fi interface is currently associated with, or wireless peer in case of the ad-hoc network.
Type:	Byte
Value Range:	0-255, default= 0.
See Also:	Associating With Selected Network ⁵¹¹ , Scanning for Wi-Fi networks ⁵⁰⁹

Details

The signal strength is expressed in 256 arbitrary levels that do not correspond to any standard measurement unit.

This property is only updated while the Wi-Fi interface is in the associated state ([wln.associationstate](#)⁵¹⁴= 1- PL_WLN_ASSOCIATED), or running an ad-hoc network ([wln.associationstate](#)⁵¹⁴= 2- PL_WLN_OWN_NETWORK). There is another read-only property -- [wln.scanresultrssl](#)⁵²⁶ -- that will contain the signal strength for the specific wireless network having been scanned for with the [wln.scan](#)⁵²⁴ method.

.Scan Method

Function:	Causes the Wi-Fi interface to commence either the search for available wireless networks or obtainment of additional information about a particular network specified by its SSID.
Syntax:	wln.scan(byref ssid as string) as accepted_rejected
Returns:	One of accepted_rejected constants: 0- ACCEPTED. 1- REJECTED.
See Also:	Scanning for Wi-Fi Networks ⁵⁰⁹ , Wln Tasks ⁵⁰⁰

Part	Description
ssid	Network name.

Details

If the ssid argument is left empty, the wln object will search for all available wireless networks. If the ssid argument is set, the wln object will obtain additional information about this network.

The scan process is a [task](#)⁵⁰⁰. As such, the wln.scan will be rejected (return 1- REJECTED) if another task is currently in progress. The task will also be rejected if the Wi-Fi interface is in the associated state ([wln.associationstate](#)⁵¹⁴= 1- PL_WLN_ASSOCIATED), is running its own network ([wln.associationstate](#)⁵¹⁴= 2- PL_WLN_OWN_NETWORK), or if the Wi-Fi hardware is not online ([wln.enabled](#)⁵¹⁹= 0- NO). The method will return 0- ACCEPTED if the task is accepted for processing.

The task is completed when the [wln.task](#)^[528] R/O property becomes 0-PL_WLN_TASK_IDLE. The [on_wln_task_complete](#)^[523] event will also be generated at that time. After the completion, the following read-only properties will be updated:

If the ssid argument was left empty, the [wln.scanresultssid](#)^[526] will contain a comma-delimited list of available wireless networks or nothing if no networks were discovered.

If the ssid argument specified a particular network, the wln.scanresultssid will contain the name of this network, or nothing if the network wasn't found. If the network was found, the [wln.scanresultbssid](#)^[525], [wln.scanresultbssmode](#)^[525], [wln.scanresultchannel](#)^[526], and [wln.scanresultrssi](#)^[526] will be updated with the information pertaining to the specified network.

.Scanresultbssid R/O Property

Function:	After a successful scan for a particular network (wln.scan ^[524] with the ssid specified) this property will contain the BSSID ("MAC address") of this network.
Type:	Dot-decimal string
Value Range:	Standard 6-byte MAC value
See Also:	Scanning for Wi-Fi Networks ^[509] , wln.scanresultbssmode ^[525] , wln.scanresultchannel ^[526] , wln.scanresultrssi ^[526] , wln.scanresultssid ^[526]

Details

This property will not be updated if the [wln.scan](#)^[524] method is invoked with its ssid argument left empty.

.Scanresultbssmode R/O Property

Function:	After a successful scan for a particular network (wln.scan ^[524] with the ssid specified) this property will contain the network mode of this network.
Type:	Enum, byte
Value Range:	0- PL_WLN_BSS_MODE_INFRASTRUCTURE: wireless network with an access point. 1- PL_WLN_BSS_MODE_ADHOC: device-to-device network without an access point.
See Also:	Scanning for Wi-Fi Networks ^[509] , wln.scanresultbssid ^[525] , wln.scanresultchannel ^[526] , wln.scanresultrssi ^[526] , wln.scanresultssid ^[526]

Details

This property will not be updated if the [wln.scan](#)^[524] method is invoked with its ssid argument left empty.

.Scanresultchannel R/O Property

Function:	After a successful scan for a particular network (wln.scan [524] with the ssid specified) this property will contain the number of the channel on which this network operates.
Type:	Byte
Value Range:	1-14
See Also:	Scanning for Wi-Fi Networks [509], wln.scanresultbssid [525], wln.scanresultbssmode [525], wln.scanresultrssi [526], wln.scanresultssid [526]

Details

This property will not be updated if the [wln.scan](#) [524] method is invoked with its ssid argument left empty.

.Scanresultrssi R/O Property

Function:	After a successful scan for a particular network (wln.scan [524] with the ssid specified) this property will contain the strength of the signal received from this network.
Type:	Byte
Value Range:	0-255
See Also:	Scanning for Wi-Fi Networks [509], wln.scanresultbssid [525], wln.scanresultbssmode [525], wln.scanresultchannel [526], wln.scanresultssid [526]

Details

This property will not be updated if the [wln.scan](#) [524] method is invoked with its ssid argument left empty.

Another read-only property -- [wln.rssi](#) [524] -- reports the signal strength of the network the Wi-Fi interface is associated with, or wireless peer in case of the ad-hoc network.

.Scanresultssid R/O Property

Function:	After the scan this property will contain a comma-delimited list of discovered networks or the name of a particular network depending on how the scan was performed.
Type:	String
Value Range:	1-32 characters
See Also:	Scanning for Wi-Fi Networks [509], wln.scanresultbssid [525], wln.scanresultbssmode [525], wln.scanresultchannel [526], wln.scanresultrssi [526], wln.scanresultssid [526]

Details

If the [wln.scan](#)^[524] method was invoked with its name argument left empty, the [wln.scanresultssid](#)^[525] will contain the list of all discovered networks. If the name argument specified a particular network and scanning found this network to be present, then this property will contain the name of this network. Naturally, this name will match the one specified by the name argument of the [wln.scan](#)^[524] method.

.Settxpower Method

Function:	Causes the Wi-Fi interface to commence the adjustment of TX power to the level specified by the level argument.
Syntax:	wln.settxpower(level as byte) as accepted_rejected
Returns:	One of accepted_rejected constants: 0- ACCEPTED. 1- REJECTED.
See Also:	Setting TX Power ^[509] , Wln Tasks ^[500]

Details

Part Description

level	Value between 4 and 15 that roughly corresponds to the transmitter's output power in dB. Attempting to specify the level<4 results in level=4; attempting to specify the level>15 results in level=15.
-------	--

Adjusting TX power is a [task](#)^[500]. As such, the wln.settxpower will be rejected (return 1- REJECTED) if another task is currently in progress. The task will also be rejected if the Wi-Fi hardware is not online ([wln.enabled](#)^[519]= 0- NO). The method will return 0- ACCEPTED if the task is accepted for processing.

The task is completed when the [wln.task](#)^[526] R/O property becomes 0- PL_WLN_TASK_IDLE. The [on_wln_task_complete](#)^[523] event will also be generated at that time.

.Setwep Method

Function:	Causes the Wi-Fi interface to commence setting new WEP mode and/or key.
Syntax:	wln.setwep(byref wepkey as string, wepmode as pl_wln_wep_modes) as accepted_rejected
Returns:	One of accepted_rejected constants: 0- ACCEPTED. 1- REJECTED.
See Also:	Setting WEP Mode and Key ^[510] , Wln Tasks ^[500]

Part Description

wep key	A string containing new WEP key. This is a "HEX strings" -- each character in the string represents one HEX digit. The string must contain 10 HEX digits for 64-bit WEP and 26 HEX digits for 128-bit WEP. Excessive digits are ignored. Missing digits are assumed to be 0.
wep mode	WEP mode to set: 0- PL_WLN_WEP_MODE_DISABLED: WEP is to be disabled. 1- PL_WLN_WEP_MODE_64: 64-bit WEP is to be used. 0- PL_WLN_WEP_MODE_128: 128-bit WEP is to be used.

Details

The WEP mode must be set prior to performing association ([wln.associate](#)⁵¹³) or starting own ad-hoc network ([wln.networkstart](#)⁵²¹).

Changing WEP mode and keys is a [task](#)⁵⁰⁰. As such, the wln.setwep will be rejected (return 1- REJECTED) if another task is currently in progress. The task will also be rejected if the Wi-Fi interface is in the associated state ([wln.associationstate](#)⁵¹⁴= 1- PL_WLN_ASSOCIATED), is running its own network ([wln.associationstate](#)⁵¹⁴= 2- PL_WLN_OWN_NETWORK), or if the Wi-Fi hardware is not online ([wln.enabled](#)⁵¹⁹= 0- NO). The method will return 0- ACCEPTED if the task is accepted for processing.

The task is completed when the [wln.task](#)⁵²⁸ R/O property becomes 0- PL_WLN_TASK_IDLE. The [on_wln_task_complete](#)⁵²³ event will also be generated at that time.

Notice that only one WEP key (wepkey1) is used by the wln object.

.Ssid Property

Obsolete.

This parameter is now supplied directly, as the argument for the [wln.associate](#)⁵¹³ method.

.Task R/O Property

Function: Indicates current wln task being executed.

Type: Enum (pl_wln_tasks, byte)

Value Range:

- 0- PL_WLN_TASK_IDLE (**default**): No task is in progress.
- 1- PL_WLN_TASK_SCAN: Scan task is in progress (initiated by [wln.scan](#)^[524]).
- 2- PL_WLN_TASK_ASSOCIATE: Association task is in progress (initiated by [wln.associate](#)^[513]).
- 3- PL_WLN_TASK_SETTXPOWER: TX power adjustment is in progress (initiated by [wln.settxpower](#)^[527]).
- 4- PL_WLN_TASK_SETWEP: WEP mode and keys setup is in progress (initiated by [wln.setwep](#)^[527]).
- 5- PL_WLN_TASK_DISASSOCIATE: Disassociation task is in progress (initiated by [wln.disassociate](#)^[518]).
- 6- PL_WLN_TASK_NETWORK_START: Ad-hoc network creation task is in progress (initiated by [wln.networkstart](#)^[521]).
- 7- PL_WLN_TASK_NETWORK_STOP: Ad-hoc network termination task is in progress (initiated by [wln.networkstop](#)^[522]).

See Also:[WIn Tasks](#)^[500]

Details

The wln object will only accept another task for execution after the previous task has been completed (wln.task= 0- PL_WLN_TASK_IDLE). Whenever a task completes, an [on_wln_task_complete](#)^[523] event is generated.

.Wepkey1 Property

Obsolete.

This parameter is now supplied directly, as the argument for the [wln.setwep](#)^[527] method.

.Wepkey2 Property

Obsolete.

Only wepkey1 is currently used by the wln object. Keys 2, 3, and 4 are never utilized.

.Wepkey3 Property

Obsolete.

Only wepkey1 is currently used by the wln object. Keys 2, 3, and 4 are never utilized.

.Wepkey4 Property

Obsolete.

Only wepkey1 is currently used by the wln object. Keys 2, 3, and 4 are never utilized.

.Wepmode Property

Obsolete.

This parameter is now supplied directly, as the argument for the [wln.setwep](#)^[527] method.

Update History (for this Manual)

29JUL2009 release

- Uncluttered platform documentation -- made these topics "common":
 - [Supported Variable Types \(T1000-based Devices\)](#)^[183];
 - [Supported Functions \(T1000-based Devices\)](#)^[183];
 - [LED Signals](#)^[184];
 - [Debug Communications](#)^[185];
 - [Project Settings Dialog](#)^[186].
- Merged EM1000 and EM1000W platform documentation under a single manual
 - [EM1000 and EM1000W Platforms](#)^[139].
- Added EM1202W platform documentation into the EM1000 platform docs, renamed the section into [EM1202 and EM1202W Platforms](#)^[149].
- Documented new platforms: [EM1206](#)^[158] and [EM1206W](#)^[158], [DS1202](#)^[166], [DS1206](#)^[174].
- Reworked [Platform Specifications](#)^[135] topic.
- Documented new [insert](#)^[196] function.
- Changes in the [wln](#)^[497] object manual:
 - Every topic was updated and edited;
 - Added [Migrating From the WA1000](#)^[498] to address the changes in the wln object operation;
 - "Configuring CS Line" renamed into [Configuring Interface Lines](#)^[507];
 - "Powering Down" renamed to [Rebooting](#)^[513];
 - "Detecting Disassociation or Powerdown" renamed into [Detecting Disassociation or Offline State](#)^[513];
 - "Enabling Port" renamed into [Applying Reset](#)^[506];
 - New [Creating Own Ad-hoc Network](#)^[512] and [Terminating Own Ad-hoc Network](#)^[512] topics;
- Changes in the [button](#)^[272] object manual:
 - Expanded the [main](#)^[272] topic;
 - Documented new [button.pressed](#)^[274] R/O property;
 - Added information about "debouncing".
- Changes in the [pat](#)^[384] object manual:
 - Documented [pat.greenmap](#)^[385] and [pat.redmap](#)^[387] properties;
 - Updated all other information in relation to the above.
- Changes in the [fd](#)^[433] object manual:
 - Updated [fd.find](#)^[464] method;
 - Documented new [fd.rename](#)^[477] method.
 - In connection with the above, renamed "Creating and Deleting Files" into

[Creating, Deleting, and Renaming Files](#)^[44], expanded topic content.

- Changes in the [sock](#)^[274] object manual:
 - Updated [sock.close](#)^[328], [sock.reset](#)^[348], and [sock.discard](#)^[331] topics -- these methods are ignored when called from within an HTML page;
 - Updated [HTTP-related Buffers](#)^[315] -- an HTTP socket can now live without the RX buffer. Also, HTTP variables of any size can now be received;
 - Reworked [Working with HTTP Variables](#)^[323] -- this is now a section; explained and documented [sock.gethttprqstring](#)^[333] and [on sock postdata](#)^[344];
 - Added "Redirection and UDP" to the [Redirecting Buffers](#)^[308] topic;
 - Documented URL substitution: new [URL Substitution](#)^[322] and [sock.urlsubstitutes](#)^[364] topics;
 - Documented the data sinking feature: new [Sinking Data](#)^[309] and [sock.sinkdata](#)^[354] topics;
 - Documented the timeout counter: expanded [Closing Connections](#)^[290] topic (see Connection Timeouts), added new [sock.toutcounter](#)^[360] topic.
- Changes in the [ser](#)^[224] object manual:
 - There is a new data sinking feature, so [Sinking Data](#)^[245] and [sock.sinkdata](#)^[354] topics were added;
 - Corrected schematic diagram (C) in the [Wiegand Mode](#)^[229] topic.

31AUG2008 release

- Documented [kp. object](#)^[484], [lcd. object](#)^[392].
- Documented [md5](#)^[201], [sha1](#)^[205], [ddstr](#)^[192], and [ddval](#)^[193] syscalls.
- Documented [sys.serialnum](#)^[221] and [sys.setserialnum](#)^[222]. Added [Serial Number](#)^[216] topic.
- Added [Using Preprocessor](#)^[71] and [Scope of Preprocessor Directives](#)^[73] topics.
- Updated [The Watch](#)^[33] topic -- documented new capabilities such as true support for arrays, expressions ("x(y)'), etc.
- Update [Project Settings](#)^[38] topic -- documented new Customize button.
- Updated [EM202](#)^[134] platform -- this platform is now used by "203" devices as well.

04AUG2008 release

- Documented [fd object](#)^[433].
- Added [Legal Information](#)^[1] topic.
- Deleted "What's New in R2" and "Migration From Version 1" topics.

10MAR2008 release

- Documented [wln object](#)^[497].
- Documented new EM1000W platform.
- Documented new [romfile.offset](#)^[378] R/O property. In connection with this, updated

the following topics: [Supported Functions \(Syscalls\)](#)^[134] (EM202/200 (-EV), DS202 platform), [Romfile Object](#)^[373].

- Documented new [sock.allowedinterfaces](#)^[327], [sock.targetinterface](#)^[359], and [sock.currentinterface](#)^[331] properties. In connection with this, also edited the following topics: [Accepting Incoming Connections](#)^[279], [Establishing Outgoing Connections](#)^[287], and [Checking Connection Status](#)^[292]. Changed information in the [Supported Objects](#)^[135] (EM202^[134] platform) topic. Updated [sock.localportlist](#)^[339], [sock.targetinterface](#)^[359] property topics. Also edited "Platform-dependent Programming Information" topics of all platforms. [EM1000](#)^[139] and [EM1202](#)^[149] platforms got new "Enum pl_sock_interfaces" topics.
 - Corrected a mistake in the [Main Parameters](#)^[268] topic ([net](#)^[267] object). The topic incorrectly stated that the Tibbo Basic application can't change the MAC address, which is, in fact, possible.
 - Correction: default value for the [net.ip](#)^[270] property is "1.0.0.1", not "127.0.0.1".
 - Corrected [net.ip](#)^[270], [net.netmask](#)^[270], [net.gatewayip](#)^[270] (details portion).
 - Correction: [EM1202](#)^[149] platform does not support RTC ([rtc](#)^[389]) object.
 - Edited [Enum pl_io_num](#)^[142] topic of the [EM1000 platform](#)^[139] manual to reflect newly supported I/O lines 49-53.
 - Added to [Understanding TCP Reconnects topic](#)^[281] (section about reconnects and HTTP). Note added also to [Sock.reconmode Property](#)^[345] topic.
 - Improved "Supported Functions" and "Supported Objects" topics for all platforms.
-

04SEP2007 release

- Extended and renamed the [Project Browser](#)^[22] topic (formerly called "Using the Project Browser"). Also made new screenshot.
 - New screenshots in the [Code Auto-completion](#)^[23] topic. Text edited slightly as well.
 - Updated the [Tooltips](#)^[24] topic, created [Supported HTML Tags](#)^[26] topic. New data concerns using HTML elements in tooltips.
 - Updated the [Watch](#)^[33] and [Scopes in Watch](#)^[37] topics -- new screenshots; the text was also edited.
 - Extended the [Constants](#)^[60] topic -- added a new section about escape sequences in string constants.
 - Updated [Language Element Icons](#)^[131] (slight changes only).
-

09AUG2007 release

- Added the [EM1202](#)^[149] platform description section.
 - Corrected [RTC Object](#)^[389] topic: should be rtc.getdata and rtc.setdata, not rtc.get and rtc.set.
 - Minor corrections in the [EM1000](#)^[139] platform description section.
-

12JUN2007 release

- [Closing Connections](#)^[290] topic contained references to sock.abort method, which does not exist. Correct method name is sock.reset.

- Expanded [Establishing Outgoing Connections](#)^[287] and [Closing Connections](#)^[290] topics. Both topics now contain "Do not forget! Connection Handling is fully asynchronous" sections.
- Added "Socket re-use after connection closing" section to the [Closing Connections](#)^[290] topic.
- New [More On the Socket's Asynchronous Nature](#)^[294] topic.

12FEB2007 release

- Updated [Adding, Removing, and Saving Files](#)^[18] topic.
- Added [Graphic File Properties Dialog](#)^[128] topic.
- Updated [Working With HTML](#)^[74] topic.
- Significantly expanded [Embedding Code Within an HTML File](#)^[76] topic -- especially important: all code fragments on the HTML page are parts of one procedure.
- Updated [Using HTTP](#)^[314], [Generating Dynamic HTML pages](#)^[320], and [Working With HTTP Variables](#)^[323] topics.

27DEC2006 release

- Added "What's new in R2" and "Migration From Version 1" topics.
- Updated [The Watch](#)^[33] -- described new functionality, provided more info on how watch works.
- [Scopes in Watch topics](#)^[37] -- provided more info on how watch works.
- Updated Using the [Project Browser](#)^[22] -- selected platform is now visible in the topmost tree node.
- Updated [Program Structure](#)^[39] -- explained that event handlers can also accept arguments.
- New [Exceptions](#)^[29] topic
- Updated [Variables and Their Types](#)^[43] -- added info about dword, long, real, float, and structures.
- Updated [Type Conversion](#)^[45] -- almost 100% new text.
- New [Type Conversion in Expressions](#)^[48] -- this section has been "under construction" for a long time.
- New [Compile-time Calculations](#)^[49] topic.
- Updated [Arrays](#)^[50] topic -- new ways to declare, etc.
- New [Structures](#)^[54] topic.
- Updated and renamed "User-defined Types" topic. Now it is called [Enumeration Types](#)^[55].
- Updated [Understanding the Scope of Variables](#)^[57] topic.
- New [Declaring Variables](#)^[60] topic.
- Updated [Introduction to Procedures](#)^[62] -- explained that event handlers can also accept arguments and can never be functions procedures.
- Updated [Dim Statement](#)^[81] topic -- new data about ways to define array variables.

- New [Type...End Type Statement](#)^[94] topic.
- Updated [Passing Arguments to Procedures](#)^[64] topic (strict byref argument match is now required).
- Updated [Goto Statement](#)^[88] -- all labels are local!
- New Supported Variable Types topics for each platform ([EM202](#)^[134], [EM1000](#)).
- Updated Platform-dependent Programming Information topics for each platform ([EM202](#)^[136], [EM1000](#)^[145]).
- EM202 platform no longer supports redirection -- [Enum pl_redir](#)^[136] topic has been updated.
- Updated Supported Functions (Syscalls) for [EM202](#)^[134] and EM1000 platforms -- some stuff in, some stuff out.
- Updated [Generating Dynamic HTML Pages](#)^[320] topic -- described changed behavior when the same code snippet has to be executed from two instances of the same HTML page being sent to the browser.
- Updated [Httpnoclose Property](#)^[335] topic -- there is a new "separator" string.
- Updated [Pat.play](#)^[386] and [Beep.play](#)^[389] topics -- now "##" means x4 speed.
- New [Sys.onsysstimerperiod Property](#)^[220] topic.
- Updated [On sys timer Event](#)^[220] topic -- to reflect that there is a new [sys.onsysstimerperiod](#)^[220] property.
- New [Sock.inconenabledmaster Property](#)^[337] topic.
- Updated Accepting Incoming Connections topic -- added material regarding [sock.inconenabledmaster](#)^[337] property.
- Updated [Stor.getdata Method](#)^[381], [Stor.setdata Method](#)^[382], [Rtc.getdata Method](#)^[390], [Rtc.setdata Method](#)^[391] topics because all four methods have been renamed.
- New [Cfloat Function](#)^[190], [ftostr Function](#)^[194], [Lbin Function](#)^[197], [Lhex Function](#)^[199], [Lstr Function](#)^[199], [Lstri Function](#)^[200], [Lval Function](#)^[200], [Strtof Function](#)^[209] topics.
- Updated [Vali Function](#)^[210] topic -- this function is no longer available since [val](#)^[210] function now works both for word (unsigned) and short (signed) conversions.
- Updated [Val Function](#)^[210] topic to reflect the fact that this function is now used both for word (unsigned) and short (signed) conversions.
- Updated [Str Function](#)^[207], [Stri Function](#)^[208], [Bin Function](#)^[190], [Hex Function](#)^[195], [Val Function](#)^[210] topics -- more accurate description and examples.
- Added "declaration" to the description of all events.
- Updated [sock.event R/O Property](#)^[332] and [sock.eventsimple R/O Property](#)^[332] topics -- these properties are not longer available.
- Updated [On sock event Event](#)^[332] topic -- this event now carries newstate and newstatesimple arguments that have replaced [sock.event](#)^[332] and [sock.eventsimple](#)^[332] R/O properties.
- Updated [Checking Connection Status](#)^[292] topic to reflect the changes made to the [on sock event](#)^[343].
- New "Split Packet" Mode of TCP Data Processing^[306], [.Splittcppackets Property](#)^[355], and [On sock tcp packet arrival Event](#)^[344] topics.
- Updated certain screenshots in several topics.
- Added Image Editor topics: [Built-in Image Editor](#)^[20], [Image Menu](#)^[119], [Image Editor Toolbar](#)^[122], [Tool Properties Toolbars](#)^[122] (+ all subtopics).
- Updated [Adding, Removing, and Saving Files](#)^[18] topic (added image editor-related info).

06JULY2006 release

- Added new platform -- [EM1000](#)^[139].
- Added "Platform revision Programming Information" topics to [EM202](#)^[134] and [EM1000](#)^[139] platform documentation.
- [Stor](#)^[380] object got new property- [stor.base](#)^[381]. Entire description of the object has been updated because of that.
- Clarification has been added to the [romfile](#)^[375] object description. This object can only access first 65534 bytes of each file, even if the actual file is larger.
- Entire new [beep](#)^[387] object has been added.
- New feature in [io](#)^[365] object -- [io.enabled](#)^[370] property was added.
- New feature in [system](#)^[212] object- see PLL Management, [sys.currentpll](#)^[218], [sys.newpll](#)^[219], [sys.resettype](#)^[222].
- New features in [serial port](#)^[224] object- support for Wiegand and clock/data interfaces. New topics include: [Three Modes of the Serial Port](#)^[225] with subtopics, [ser.mode](#)^[255], and [ser.autoclose](#)^[248]. A lot of other topics have been changed- too many to list here.
- Change in [sys.buffalloc](#)^[217] behavior: now if the serial port (socket) to which the buffer belongs is not closed (idle) the buffer size will remain unchanged. This affects [ser.rxbuffrq](#)^[262], [ser.txbuffrq](#)^[265], [sock.rxbuffrq](#)^[350], [sock.txbuffrq](#)^[362], [sock.tx2buffrq](#)^[361], [sock.cmdbuffrq](#)^[329], [sock.rplbuffrq](#)^[349], [sock.varbuffrq](#)^[365].
- Corrected errors in the [Enum pl io num](#)^[136] (pin descriptions were wrong- RTS, CTS, DTR, and DSR lines were shown at incorrect positions).
- Corrected [ser.txlen](#)^[267], [ser.txfree](#)^[266], [sock.txlen](#)^[364], [sock.txfree](#)^[363] property descriptions. These properties *do not* take into account uncommitted data in the TX buffer (it was stated otherwise previously). Consequently these topics were also edited: [Buffer Memory Status](#)^[240], [TX and RX Buffer Memory Status](#)^[300], [Ser.notifysent](#)^[257], [on ser data sent](#)^[258], [sock.notifysent](#)^[340], [on sock data sent](#)^[342], [ser.setdata](#)^[264], and [sock.setdata](#)^[353] have been amended accordingly.
- Corrected mistakes related to date/time conversion functions- [date](#)^[191] function was erroneously documented as "day" function, [weekday](#)^[211] function description was missing altogether. Topics of other date/time related functions- [year](#)^[211], [month](#)^[204], [daycount](#)^[192], [hours](#)^[195], [minutes](#)^[203], and [mincount](#)^[203] were slightly corrected.

08MAY2006 release

- Corrected errors in [io.Num Property](#)^[372] and [io.State Property](#)^[375]

08MAR2006 release

- Updated [Preparing Your Hardware](#)^[9] with the network upgrade procedure
- Updated [Starting a New Project](#)^[10]
- Updated [Making, Uploading and Running an Executable Binary](#)^[26]
- Updated [Project Menu](#)^[117] with new entry description for Device Explorer
- Updated [Debug Toolbar](#)^[121] with new button description for Device Explorer
- Updated and expanded Device Explorer
- Added new functions: [Day Function](#)^[191], [Daycount Function](#)^[192], [Hours Function](#)^[195],

[Mincount Function](#)^[203], [Minutes Function](#)^[203], [Month Function](#)^[204], [Year Function](#)^[211]

11JAN2006 release

- Improved indexes -- better context search.
 - Added [L1008](#)^[112], [L1009](#)^[112]
-

02JAN2006 release

- Initial release of manual.

Index

- - -
- Operator 99

- & -

&b 40
&h 40

- * -

* Operator 99

- / -

/ Operator 99

- + -

+ Operator 99

- = -

= Operator 99

- A -

Abort 28
Accessing a Value Within an Array 50
active opens 287
Actively closing TCP connections 290
Actively closing UDP connections 290
add custom comments 24
Add File 18
AND Operator 99
arrays 50
 watching 33
asc 189
asynchronous operation 224

- B -

BASIC code snippet in HTTP file 318
BASIC files 15
baudrate 224

baudrate property 236
Beep 387
beep.divider 388
beep.play 389
Beeper 387
bin 190
blocking code 301
blue line 31
boolean 43
Break 28
breakpoint 30
Broadcast 281
buffer memory 213
buffer overruns 244, 307
buffer redirection 308
buffer shorting 224, 308
buffer sizes 224
buffers 213
Button 272
button.time 274
Buzz 10
Buzzer 387
By Reference 64
By Value 64
byte 43

- C -

C1001	101
C1002	101
C1003	102
C1004	102
C1005	102
C1006	103
C1007	103
C1008	104
C1009	104
C1010	104
C1011	105
C1012	105
C1013	106
C1014	106
C1015	106
C1016	107
C1017	107
C1018	107
C1019	108
C1020	108
C1021	108

C1022 109
 C1023 109
 C1024 110
 call stack 31, 128
 case 91
 Case Sensitive 40
 cfloat 190
 char 43
 chr 191
 CMD buffer 276, 310
 overruns 311
 Code hinting 24
 Code Profiling 37
 code-completion 23
 Colons 40
 Comments 40
 Communication in progress 28
 Communication problem 28
 Compilation Unit 131
 Compiler 131
 connections close automatically 290
 const 60, 79
 Constants 60
 In different bases 40
 Construct 132
 Conversion 45
 cross-debugging 28, 132
 Ctrl+Shift+space 24
 Ctrl+space 23
 CTS line 225
 CTS/RTS flow control 224
 custom comments for tooltips 24

- D -

data overrun detection 224
 date 191
 daycount 192
 ddstr 192
 ddval 193
 Debug Mode 27, 114
 Debug version 15
 Decision Structures 67
 Declares 79
 Declaring Procedures 62
 Declaring Variables 57
 default gateway 267
 dim 81
 direction control via RTS 224

do 82
 doevents 68, 82, 320
 Double Quote Marks 40
 DS202 134
 dynamic HTML 74, 315, 320

- E -

Edit Mode 114
 EEPROM 380
 else 89
 elseif 89
 EM1000 139
 EM1000-EV 139
 EM1202 149
 EM1202-EV 149
 EM200 134
 EM202 134
 EM202-EV 134
 end 85
 end if 89
 end select 91
 end sub 93
 enum 55, 84
 enumeration types 43
 escape character 309
 escape sequence 76
 escape sequences 224
 Ethernet communications 268
 events 8
 event handlers 11, 39
 exit 85
 exit do 85
 exit for 85
 exit function 85
 exit sub 85
 exit while 85

- F -

F5 14, 26
 F7 26
 F9 30
 fd.availableflashspace 457
 fd.buffernum 458
 fd.capacity 458
 fd.checksum 459
 fd.close 460
 fd.copyfirmware 460

fd.create 462
 fd.cutfromtop 461
 fd.delete 463
 fd.filenum 463
 fd.fileopened 464
 fd.filesize 464
 fd.find 464
 fd.flush 466
 fd.format 467
 fd.getattributes 467
 fd.getbuffer 468
 fd.getdata 469
 fd.getfreespace 470
 fd.getnextdirmember 470
 fd.getnumfiles 471
 fd.getsector 472
 fd.laststatus 472
 fd.maxopenedfiles 473
 fd.maxstoredfiles 474
 fd.mount 474
 fd.numservicesectors 475
 fd.open 475
 fd.pointer 476
 fd.ready 476
 fd.resetdirpointer 478
 fd.sector 478
 fd.setattributes 478
 fd.setbuffer 479
 fd.setdata 480
 fd.setfilesize 481
 fd.setpointer 482
 fd.setsector 483
 fd.totalsize 483
 file pointer 375
 firewall 9
 firmware file 9
 for 86
 form 323
 freeze 301
 ftosr 194
 Full duplex 224
 Function 87
 Function Procedures 62

- G -

GIF 314
 Global Scope 57
 Global Variables 39

goto 88
 graceful disconnect 278
 green LED 384
 green status 28
 GUI 113
- H -
 half duplex 224
 halt 217
 Handling RX buffer overruns 244
 Header files 15
 hex 195
 hours 195
 hover your mouse 24
 HTML 74, 314
 dynamic content 74
 dynamic data 320
 Dynamic pages 315
 files 15
 form 323
 Pages 74
 Scope 57
 HTTP 277
 mode 314
 server 314
 Variables 323

- I -

icons 131
 Identifier 132
 Identifiers 42
 if statement 89
 illegal characters 320
 Inband commands 309
 Inband message 309
 Inband replies 313
 Include 39
 include "filename" 90
 incoming connections mode 279
 instr 197
 integer 43
 Integers 43
 IO Object 365
 io.enabled 370
 io.intenabled 370
 io.intnum 371
 io.invert 371
 io.lineget 371

io.lineset 372
 io.num 372
 io.portenabled 373
 io.portget 373
 io.portnum 374
 io.portset 374
 io.portstate 375
 io.state 375

- J -

JPG 314
 Jump to Cursor 32

- K -

Keyword 132
 kp.autodisablecodes 492
 kp.enabled 492
 kp.longpressdelay 493
 kp.longreleasedelay 493
 kp.pressdelay 495
 kp.releasedelay 495
 kp.repeatdelay 496
 kp.returnlinesmapping 496
 kp.scanlinesmapping 497

- L -

L1001 110
 L1002 110
 L1003 111
 L1004 111
 L1005 111
 L1006 111
 L1007 112
 L1008 112
 L1009 112
 label 88, 132
 lbin 197
 lcd.backcolor 414
 lcd.bitsperpixel 414
 lcd.bluebits 415
 lcd.bmp 415
 lcd.enabled 416
 lcd.error 417
 lcd.fill 417
 lcd.filledrectangle 418
 lcd.fontheight 418

lcd.fontpixelpacking 419
 lcd.forecolor 419
 lcd.getprintwidth 420
 lcd.greenbits 420
 lcd.height 421
 lcd.horline 421
 lcd.inverted 422
 lcd.iomapping 422
 lcd.line 422
 lcd.linewidth 423
 lcd.lock 423
 lcd.lockcount 424
 lcd.panelype 424
 lcd.pixelpacking 425
 lcd.print 425
 lcd.printaligned 426
 lcd.rectangle 427
 lcd.redbits 427
 lcd.rotated 428
 lcd.setfont 428
 lcd.setpixel 429
 lcd.textalignment 429
 lcd.texthorizontalspacing 430
 lcd.textorientation 430
 lcd.textverticalspacing 431
 lcd.unlock 431
 lcd.verline 432
 lcd.width 432

LED 384
 left 198
 len 198
 lhex 199
 Linker 132
 link-level broadcasts 281
 Listening ports 279
 Local Scope 57
 loop 82
 Loop structures 68
 loopback 301
 lstr 199
 lstri 200
 lval 200

- M -

main window 114
 Master Process 7
 md5 201
 memory allocation 213, 239, 297

memory capacity 239
 menu 115
 messages embedded within the TCP data stream 309
 mid 202
 mincount 203
 minimalistic 77
 minutes 203
 MOD Operator 99
 month 204
 more than one serial port 234
 Multi-Dimensional Arrays 50
 Multiple Sockets 286

- N -

Net object 267
 net.failure 271
 net.gatewayip 270
 net.ip 270
 net.linkstate 271
 net.mac 269
 net.netmask 270
 new project 10
 next 86
 No Communication 28
 non-blocking operation 240
 non-HTTP and HTTP processing on the same socket 317
 NOT Operator 99

- O -

Objects 8, 78
 on_beep 388
 on_button_pressed 273
 on_button_released 273
 on_io_int 373
 on_kp 494
 on_kp_overflow 494
 on_net_link_change 271
 on_net_overrun 272
 on_pat 386
 on_ser_data_arrival 258
 On_ser_data_arrival Event 241
 on_ser_data_sent 258
 on_ser_esc 259
 on_ser_overrun 259
 on_sock_data_arrival 342
 on_sock_data_sent 342

on_sock_event 343
 on_sock_inband 343
 on_sock_overrun 343
 on_sock_tcp_packet_arrival 344
 on_sys_init 212, 220
 on_sys_timer 220
 on_wln_event 522
 on_wln_task_complete 523
 OR Operator 99

- P -

parity 224
 passive open 278
 Passive TCP connection termination 290
 pat.play 386
 Pause 28
 P-Code 132
 Philosophy 4
 pl_io_num 136
 pl_redir 136
 PL_SST_CL_ARESET_CMD 292
 PL_SSTS_CLOSED 292
 Platform Functions 78
 point-to-point 278
 polling 241, 301
 Port Selection 234
 port switchover 283
 program pointer 30
 project 15
 Project file 15
 Project pane 129
 Project tree 18

- Q -

queue 7

- R -

RAM 66
 random 205
 read data from EEPROM 380
 Real-time Clock 389
 reboot your device manually 26
 Receiving Data 240, 298, 299
 receiving data with UDP 303
 reconmode 281
 reconnects 281, 283

Recursion 62
red dot 30
red LED 384
red status 28
Release Mode 27
Remove All Breakpoints 30
Remove File 18
Resource files 15, 20
Restart 26
right 205
Romfile Object 375
romfile.find 377
romfile.getdata 378
romfile.offset 378
romfile.open 379
romfile.pointer 379
romfile.size 379
RPL buffer 276, 310, 313
rtc.get 390
rtc.running 391
rtc.set 391
RTS line 225
Run 28
Run to Cursor 32
RX buffer 225, 240, 276, 300, 315
RX buffer overruns 307
RX line 225

- S -

sandbox 7, 133
scope 57
select case 91
send UDP broadcasts 288
Sending data 243, 298, 299, 304
ser.autoclose 248
ser.baudrate 248
ser.bits 249
ser.ctsmap 249
ser.dircontrol 250
ser.div9600 250
ser.enabled 251
ser.escchar 251
ser.esctype 251
ser.flowcontrol 253
ser.getdata 253
ser.interchardelay 254
ser.interface 255
ser.mode 255

ser.newtxlen 256
ser.notifysent 257
ser.num 257
ser.numofports 258
ser.parity 260
ser.redir 260
ser.rtsmap 261
ser.rxbuffrq 262
ser.rxbuffsize 262
ser.rxclear 263
ser.rxlen 263
ser.send 264
ser.setdata 264
ser.txbuffrq 265
ser.txbuffsize 266
ser.txclear 266
ser.txfree 266
ser.txlen 267
serial port 225
serial port object 224
Serial Settings 236
set the socket for HTTP 317
Settings 38
sha1 205
short 43
Single Quote Marks 40
Sock Object 274
sock.acceptbcast 327
sock.allowedinterfaces 327
sock.bcast 328
sock.close 328
sock.cmdbuffrq 329
sock.cmdlen 329
sock.connect 330
sock.connectiontout 330
sock.currentinterface 331
sock.discard 331
sock.endchar 331
sock.escchar 332
sock.event 332
sock.eventsimple 332
sock.getdata 333
sock.getinband 334
sock.httpmode 334
sock.httpnoclose 335
sock.httpportlist 335
sock.httprqstring 336
sock.inbandcommands 337
sock.inconenabledmaster 337

sock.inconmode 338
 sock.localport 338
 sock.localportlist 339
 sock.newtxlen 339
 sock.nextpacket 303, 339
 sock.notifysent 340
 sock.num 340
 sock.numofsock 341
 sock.outport 341
 sock.protocol 345
 sock.reconmode 345
 sock.redir 346
 sock.remoteip 347
 sock.remotemac 348
 sock.remoteport 348
 sock.reset 348
 sock.rplbuffrq 349
 sock.rplfree 350
 sock.rpllen 350
 sock.rxbuffrq 350
 sock.rxbuffsize 351
 sock.rxclear 351
 sock.rxlen 352
 sock.rxpathetlen 352
 sock.send 353
 sock.setdata 353
 sock.setsendinband 354
 sock.splittcpackets 355
 sock.state 292, 355
 sock.statesimple 292, 358
 sock.targetbcast 359
 sock.targetinterface 359
 sock.targetip 359
 sock.targetport 360
 sock.tx2buffrq 361
 sock.tx2len 362
 sock.txbuffrq 362
 sock.txbuffsize 363
 sock.txclear 363
 sock.txfree 363
 sock.txlen 364
 sock.varbuffrq 365
 socket
 automatic switching 276
 stack pointer 31
 state 28
 Statements 79
 status bar 126
 status messages 28
 stepping 32
 stor.base 381
 stor.get 381
 stor.set 382
 stor.size 383
 str 207
 strgen 207
 stri 208
 string 43
 strsum 209
 strtod 209
 sub 93
 Sub Procedures 62
 SYN-SYN-ACK 287
 Sys Object 212
 sys.buffalloc 217
 sys.currentpll 218
 sys.freebuffpages 218
 sys.halt 219
 sys.newpll 219
 sys.onsysstimerperiod 220
 sys.reboot 221
 sys.resettype 222
 sys.runmode 221
 sys.serialnum 221
 sys.setserialnum 222
 sys.timercount 223
 sys.totalbuffpages 223
 sys.version 223
 Syscall 132
 system requirements 113

- T -

Target 133
 tbh 15
 tbs 15
 TCP 278
 Templates 17
 terms 131
 Tibbo Basic code within an HTML file 76
 Timekeeping 389
 timeouts 290
 timer 37, 214
 toolbars 120
 tooltip 24
 total capacity of the buffer 240
 tpr 15
 tree 66

TX buffer 225, 240, 276, 300, 315
TX buffer overruns 244, 307
TX line 225
TX2 buffer 276, 310
TXT 314

- U -

UDP "connections" 278
UDP broadcasts
 accept 281
 send 288
until 82
Upload 26

- V -

val 210
vali 210
VAR buffer 276, 315
Variable Types For Arrays 50
Virtual Machine 7, 28, 133

- W -

watch 33
watching arrays 33
weekday 211
wend 95
while 82, 95
who can connect 279
window 114
wln.associate 513
wln.associationstate 514
wln.boot 515
wln.bssmode 515
wln.buffrq 516
wln.bufffsize 516
wln.csmap 517
wln.defaultibsschannel 517
wln.disassociate 518
wln.domain 518
wln.enabled 519
wln.gatewayip 520
wln.ip 520
wln.mac 520
wln.netmask 521
wln.rssi 524
wln.scan 524

wln.scanresultbssid 525
wln.scanresultbssmode 525
wln.scanresultchannel 526
wln.scanresultrssi 526
wln.scanresultssid 526
wln.settxpower 527
wln.setwep 527
wln.ssid 528
wln.task 528
wln.wepkey1 529
wln.wepkey2 529
wln.wepkey3 529
wln.wepkey4 529
wln.wemode 530
word 43
word length 224
write data to EEPROM 380

- X -

XOR Operator 99

- Y -

year 211
yellow line 30
yellow status 28