IPL-ST7 inDART Programming Library

Programmer's Manual

SofTec Microsystems

E-mail (general information): info@softecmicro.com

E-mail (technical support): support@softecmicro.com

Web: http://www.softecmicro.com

Contents

Overview

Introduction

This documentation deals with low-level interfacing between user written PC applications and the inDART-ST7 Series In-Circuit Debuggers/Programmers. This section assumes you have already read the instrument's user's manual and got acquainted with the instrument. All of the examples provided in this documentation are written in C, unless otherwise reported.

The IPL-ST7 DLL

Dynamic-link libraries (DLL) are modules that contain functions and data. A DLL is loaded at run time by its calling modules (.exe or .dll). When a DLL is loaded, it is mapped into the address space of the calling process.

The IPL-ST7 Programming Library is a DLL which includes all of the low-level functions that allow you to set up the instrument and perform, from within your own application, most of the programming commands and functions of the DataBlaze user interface.

The IPL-ST7 Programming Library contains C written routines, and can be used to interface the instrument from within, for example, a Microsoft Visual C or Visual Basic application, as well as any other programming language that supports the DLL mechanism. For details on how to call DLL functions from within your application, please refer to the your programming language's documentation.

Installation

Before to start working with the IPL-ST7 Programming Library, you must set up your system with all the required files and drivers. You must perform the following installation steps:

1. Copy the IPL-ST7 Programming Library files to the hard disk.

2. Install the inDART communication driver (LPT or USB driver, depending on the inDART model).
3. Install the SmartKey driver (the SmartKey is the provided USB dongle which must be connected to a USB port in order for the IPL-ST7 Programming Library to work—it's an anti-piracy protection system).
4. Copy the `IPL_ST7.dll` file and the appropriate `inDART_ST7.dll` file to your application's executable directory.

1. Copying the Files to the Hard Disk

The IPL-ST7 Programming Library comes as a zipped archive. You can unzip it to any hard disk location. After unzipping, the IPL-ST7 directory structure will be the following:

Directory	Description
DLLs\inDART_ST7	Contains the inDART_ST7.dll file for inDART-ST7 models (inDART-ST7C, inDART-ST7F, inDART-ST72F264, inDART-ST7FLITE0). This file must be copied to your application's directory.
DLLs\inDART_STX	Contains the inDART_ST7.dll file for inDART-STX. This file must be copied to your application's directory.
DLLs\IPL_ST7	Contains the IPL_ST7.dll file. This file must be copied to your application's directory. Also contains .lib and .h files for Visual C applications.
Docs	Documentation.
Drivers\LPT	Contains the driver needed by LPT-based inDARTs in order to communicate with the PC. Note: this driver is automatically installed by the inDART system software that came with the instrument. If you choose not to install the inDART system software, you can manually install this driver following the instructions provided in the readme.txt file in this directory.
Drivers\USB\inDART-STX	Contains the driver needed by the USB-based inDART-STX in order to communicate with the PC. Note: this driver is automatically installed by the inDART system software that came with the instrument. If you choose not to install the inDART system software, you can manually install this driver following the instructions provided in the readme.txt file in this directory.
Drivers\USB\inDART-ST7F	Contains the driver needed by the USB-based inDART-ST7F in order to communicate with the PC. Note: this driver is automatically installed by the inDART system software that came with the instrument. If you choose not to install the inDART system software, you can manually install this driver following the instructions provided in the readme.txt file in this directory.
Examples	Contains a Visual C sample project which uses the IPL-ST7 programming library.
SmartKey	Contains the driver for the SmartKey dongle, which must be connected to a USB port in order for the IPL-ST7 programming interface to work.

Interface Library Contents

2. Installing the inDART Communication Driver

Depending on the inDART model you are using, you must install either the LPT driver or the USB driver.

Note: *if you ran the inDART system software that came with the instrument, the appropriate communication driver is already present in your system. If you haven't done so, and you wish to install the driver yourself, follow the instructions below.*

To install the LPT driver (for LPT-based inDART models):

1. If you are working under Windows 9x/Me, just copy the file `TVicLPT.vxd` from the `Drivers\LPT` directory to the `<WINDOWS>\SYSTEM` directory.
2. If you are working under Windows NT/2000/XP, you must do the following:

    a. Log in as Administrator;

    b. Copy the file `TVicLPT.sys` from the `Drivers\LPT` directory to the `<WINDOWS>\SYSTEM32\DRIVERS` directory;

    c. Create the following keys in the Registry:
    `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\TVicLPT`, Key = "`ErrorControl`", Value = `0x00000001`, Type = `DWORD`
    `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\TVicLPT`, Key = "`Type`", Value = `0x00000001`, Type = `DWORD`
    `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\TVicLPT`, Key = "`Start`", Value = `0x00000002`, Type = `DWORD`
    `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\TVicLPT`, Key = "`Group`", Value = "`Extended Base`", Type = `String`
    `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\TVicLPT`, Key = "`Parameters`", Value = "", Type = `String`

    d. Reboot the PC.

To install the USB driver for the inDART-ST7F model:

1. Copy the `st7fusb.inf` file from the `Drivers\USB\inDART-ST7F` directory to the `<WINDOWS>\Inf` directory.
2. Copy the `sftst7f.sys` and `sftst7fl.sys` files from the `Drivers\USB\inDART-ST7F` directory to the `<WINDOWS>\System32\Drivers` directory.

To install the USB driver for the inDART-STX model:

1. Copy the `stxusb.inf` file from the `Drivers\USB\inDART-STX` directory to the `<WINDOWS>\Inf` directory.
2. Copy the `sftstx.sys` file from the `Drivers\USB\inDART-STX` directory to the `<WINDOWS>\System32\Drivers` directory.

3. Installing the SmartKey Driver

To install the SmartKey driver, please follow these steps:

1. Plug the SmartKey USB in a USB port.
2. When Windows asks for the driver, select the `SmartKey` directory (which contains the `skeyusb.inf` file) and Windows will automatically install the driver.

4. Copying the required DLLs to your application's executable directory

Finally, you must copy the `IPL_ST7.dll` file and the appropriate `inDART_ST7.dll` file to your application's executable directory.

1. Copy the `IPL_ST7.dll` file from the `DLL\IPL_ST7` directory to your application's executable directory.
2. There are two version of the `inDART_ST7.dll` file: one for inDART-ST7 models (inDART-ST7C, inDART-ST7F, inDART-ST72F264, inDART-ST7FLITE0) and one for the inDART-STX model.
   Depending on which inDART model you are working with, you must copy the `inDART_ST7.dll` file (either from the `DLLs\inDART_ST7` directory or from the `DLLs\inDART_STX` directory) to your application's executable directory.

Programming Library Reference

Using the Interface Library Functions

When you control inDART within your own application, you will typically follow the steps indicated below:

1. **Initialize the instrument.**
   To communicate with inDART you need to open a parallel port or USB resource and initialize the inDART board with target device information. This initialization procedure must be done every time the instrument is powered on. To initialize the instrument, call the `IPL_OpenCommunication` function.
2. **Program.**
   Once the instrument has been set up, you can begin programming. Each programming operation (blank check, erase, read, program, etc.) must be called within a `IPL_StartProgramming` and `IPL_EndProgramming` block.
3. **Close the communication with the instrument.**
   Closing the communication with the instrument frees the parallel port or USB resource used during communication.

Return Values of the Programming Library Functions

Most of the IPL-ST7 Programming Library functions return a BOOL value which indicates whether the function has been successfully executed (return value = `TRUE`) or not (return value = `FALSE`). In the latter case it is possible to get extended error information by calling the function `IPL_GetErrorMessage`:

```
void IPL_GetErrorMessage (char *msg);
```

The `msg` parameter will be filled with a text message explaining the cause of the problem.

Function Reference

IPL_BlankCheck

Include file:

```
#include "IPL_ST7.h"
```

Function prototype:

```
BOOL IPL_BlankCheck
  (int mem_type,
  unsigned long start_addr,
  unsigned long len,
  BOOL *result,
  IPL_ProgressProc *callback);
```

Parameters:

`mem_type`:	the type of memory affected. Can be one of the following values: `IPL_MEMORY_CODE` `IPL_MEMORY_DATA` `IPL_MEMORY_OPTION`
`start_addr`:	the start address. When blank checking the Option Bytes, set this parameter to `0`.
`len`:	the number of bytes (starting from `start_addr`) to blank check. When blank checking the Option Bytes, this parameter is `2` for devices with 2 Option Bytes, and `1` for devices with 1 Option Byte.
`result`:	returns `TRUE` if the checked memory if blank, `FALSE` otherwise.
`callback`:	specifies the address of a callback function which will get the progress (a value from 0 to 100) of the blank check operation. You can use this feature to display a progress bar while the target's memory is being blank checked. If this feature is not used, the `callback` parameter must be `NULL`. The callback function prototype must be defined as follows:

```
void IPL_ProgressProc (unsigned long);
```

Return value:

`TRUE`:	the function was successful.
`FALSE`:	an error occurred. Call the `IPL_GetErrorMessage` function to get extended error information.

Description:

Checks if a memory portion of the target device is blank.

Note: this function must be called within an `IPL_StartProgramming`/ `IPL_EndProgramming` block.

IPL_CheckDeviceProtection

Include file:

```
#include “IPL_ST7.h”
```

Function prototype:

```
BOOL IPL_CheckDeviceProtection (BOOL *dev_protected);
```

Return value:

`TRUE`:	the function was successful.
`FALSE`:	an error occurred. Call the `IPL_GetErrorMessage` function to get extended error information.

Description:

Returns (in the `dev_protected` variable) whether the code memory of the target device is protected or not.

Note: this function must be called within an `IPL_StartProgramming`/ `IPL_EndProgramming` block.

IPL_CloseCommunication

Include file:

```
#include “IPL_ST7.h”
```

Function prototype:

```
BOOL IPL_CloseCommunication (void);
```

Return value:

`TRUE`:	the function was successful.
`FALSE`:	an error occurred. Call the `IPL_GetErrorMessage` function to get extended error information.

Description:

Closes the communication with the instrument and frees the parallel port or USB resource used during communication.

IPL_EndProgramming

Include file:

```
#include "IPL_ST7.h"
```

Function prototype:

```
BOOL IPL_EndProgramming (void);
```

Return value:

`TRUE`:	the function was successful.
`FALSE`:	an error occurred. Call the `IPL_GetErrorMessage` function to get extended error information.

Description:

Each programming operation (blank check, erase, read, program, etc.) must be called within a `IPL_StartProgramming` and `IPL_EndProgramming` block. The `IPL_EndProgramming` function signals the inDART board not to expect any more programming operations, and turns off the "BUSY" LED (if present on the inDART board).

An user application can have as many `IPL_StartProgramming`/ `IPL_EndProgramming` blocks as desired, as long as every programming operation is called within them.

IPL_Erase

Include file:

```
#include "IPL_ST7.h"
```

Function prototype:

```
BOOL IPL_Erase
(int mem_type,
unsigned long start_addr,
unsigned long len,
BOOL *result,
IPL_ProgressProc *callback);
```

Parameters:

`mem_type`:	the type of memory affected. Can be one of the following values:
	`IPL_MEMORY_CODE`
	`IPL_MEMORY_DATA`
	`IPL_MEMORY_OPTION`

`start_addr`:	the start address. When erasing the Option Bytes, set this parameter to `0`.
`len`:	the number of bytes (starting from `start_addr`) to erase. When erasing the Option Bytes, this parameter is `2` for devices with 2 Option Bytes, and `1` for devices with 1 Option Byte.
`result`:	returns `TRUE` if the erasing was successful, `FALSE` otherwise.
`callback`:	specifies the address of a callback function which will get the progress (a value from 0 to 100) of the erasing operation. You can use this feature to display a progress bar while the target's memory is being erased. If this feature is not used, the `callback` parameter must be `NULL`. The callback function prototype must be defined as follows: `void IPL_ProgressProc (unsigned long);`

Return value:

`TRUE`:	the function was successful.
`FALSE`:	an error occurred. Call the `IPL_GetErrorMessage` function to get extended error information.

Description:

Erases a portion of the target device's memory.

Note: this function must be called within an `IPL_StartProgramming`/ `IPL_EndProgramming` block.

IPL_GetInfo

Include file:

```
#include “IPL_ST7.h”
```

Function prototype:

```
BOOL IPL_GetInfo (char *info);
```

Return value:

`TRUE`:	the function was successful.
`FALSE`:	an error occurred. Call the `IPL_GetErrorMessage` function to get extended error information.

Description:

Returns a string containing the version of the IPL-ST7 Programming Library and the version of the underlying `inDART_ST7.dll` file in use.

IPL_OpenCommunication

Include file:

```
#include “IPL_ST7.h”
```

Function prototype:

```
BOOL IPL_OpenCommunication
  (int comm_channel,
  const char *hw_model,
  const char *device_code,
  unsigned long reset_rise_time,
  BOOL use_option_bytes,
  int ec_freq);
```

Parameters:

`comm_channel`:	the communication channel between the inDART board and the PC. Valid values are:
	`IPL_USB`
	`IPL_LPT1`
	`IPL_LPT2`
	`IPL_LPT3`
	`IPL_LPT4`

`hw_model`:	a string defining the inDART board model. Valid values are: `IPL_INDART_ST72F264` `IPL_INDART_ST7C` `IPL_INDART_ST7F` `IPL_INDART_ST7FLITE0` `IPL_INDART_STX`
`device_code`:	a string representing the exact code of the target device, as listed by the DataBlaze programming utility.
`reset_rise_time`:	is the target RESET rise time. Depending on the specific inDART board model and target device you are working with, this parameter is used by the IPL-ST7 Programming Library to correctly initialize the instrument. For further details, please refer to the appropriate inDART user's manual.
`use_option_bytes`:	specifies how the instrument will enter the ICC mode. Depending on the specific inDART board model and target device you are working with, this parameter is used by the IPL-ST7 Programming Library to correctly initialize the instrument. For further details, please refer to the appropriate inDART user's manual.
`ec_freq`:	sets the embedded command frequency (in MHz). Depending on the specific inDART board model and target device you are working with, this parameter is used by the IPL-ST7 Programming Library to correctly initialize the instrument. For further details, please refer to the appropriate inDART user's manual.

Return value:

`TRUE`:	the function was successful.
`FALSE`:	an error occurred. Call the `IPL_GetErrorMessage` function to get extended error information.

Description:

Opens a parallel port or USB resource and initializes the inDART board with target device information. This initialization procedure must be done every time the instrument is powered on, and before calling any other function that communicates with the instrument.

IPL_Program

Include file:

```
#include “IPL_ST7.h”
```

Function prototype:

```
BOOL IPL_Program
(int mem_type,
unsigned long start_addr,
unsigned long len,
unsigned char *mem,
BOOL *result,
IPL_ProgressProc *callback);
```

Parameters:

`mem_type`:	the type of memory affected. Can be one of the following values: `IPL_MEMORY_CODE` `IPL_MEMORY_DATA` `IPL_MEMORY_OPTION`
`start_addr`:	the start address. When programming the Option Bytes, set this parameter to `0`.
`len`:	the number of bytes (starting from `start_addr`) to program. When programming the Option Bytes, this parameter is `2` for devices with 2 Option Bytes, and `1` for devices with 1 Option Byte.
`mem`:	the pointer to the first byte of the buffer containing the data to be programmed. When programming the Option Bytes, `mem` points to the first Option Byte.
`result`:	returns `TRUE` if the programming was successful, `FALSE` otherwise.
`callback`:	specifies the address of a callback function which will get the progress (a value from 0 to 100) of the programming operation. You can use this feature to display a progress bar while the target’s memory is being programmed. If this feature is not used, the

callback parameter must be NULL. The callback function prototype must be defined as follows:

```
void IPL_ProgressProc (unsigned long);
```

Return value:

TRUE:	the function was successful.
FALSE:	an error occurred. Call the IPL_GetErrorMessage function to get extended error information.

Description:

Programs (writes) a portion of the target device's memory.

Note: this function must be called within an IPL_StartProgramming/ IPL_EndProgramming block.

IPL_Read

Include file:

```
#include "IPL_ST7.h"
```

Function prototype:

```
BOOL IPL_Read
(int mem_type,
unsigned long start_addr,
unsigned long len,
unsigned char *mem,
BOOL *result,
IPL_ProgressProc *callback);
```

Parameters:

mem_type:	the type of memory affected. Can be one of the following values: IPL_MEMORY_CODE IPL_MEMORY_DATA IPL_MEMORY_OPTION
start_addr:	the start address. When reading the Option Bytes, set this parameter to 0.

`len`:	the number of bytes (starting from `start_addr`) to read. When reading the Option Bytes, this parameter is `2` for devices with 2 Option Bytes, and `1` for devices with 1 Option Byte.
`mem`:	the pointer to the first byte of the buffer which will receive the read data. When reading the Option Bytes, `mem` points to the first Option Byte.
`result`:	returns `TRUE` if the reading was successful, `FALSE` otherwise.
`callback`:	specifies the address of a callback function which will get the progress (a value from 0 to 100) of the reading operation. You can use this feature to display a progress bar while the target's memory contents are being read. If this feature is not used, the `callback` parameter must be `NULL`. The callback function prototype must be defined as follows:

```
void IPL_ProgressProc (unsigned long);
```

Return value:

`TRUE`:	the function was successful.
`FALSE`:	an error occurred. Call the `IPL_GetErrorMessage` function to get extended error information.

Description:

Reads a portion of the target device's memory.

Note: this function must be called within an `IPL_StartProgramming`/ `IPL_EndProgramming` block.

IPL_Run

Include file:

```
#include "IPL_ST7.h"
```

Function prototype:

```
BOOL IPL_Run (void);
```

Return value:

`TRUE`:	the function was successful.

`FALSE`:	an error occurred. Call the `IPL_GetErrorMessage` function to get extended error information.

Description:

Resets the target device and restarts the microcontroller in user mode, thus fetching and executing the programmed user program.

Note: this function must be called within an `IPL_StartProgramming`/ `IPL_EndProgramming` block.

IPL_StartProgramming

Include file:

```
#include "IPL_ST7.h"
```

Function prototype:

```
BOOL IPL_StartProgramming (void);
```

Return value:

`TRUE`:	the function was successful.
`FALSE`:	an error occurred. Call the `IPL_GetErrorMessage` function to get extended error information.

Description:

Each programming operation (blank check, erase, read, program, etc.) must be called within a `IPL_StartProgramming` and `IPL_EndProgramming` block. The `IPL_StartProgramming` function prepares the inDART board to perform programming operations, and turns on the "BUSY" LED (if present on the inDART board).

An user application can have as many `IPL_StartProgramming`/ `IPL_EndProgramming` blocks as desired, as long as every programming operation is called within them.