STVP programming toolkit

Introduction

The STVP programming toolkit provides a set of C++ source and header files that can be used to design a customized programming application for any ST programming hardware that is supported by STVP.

The provided files contain the source code for functions that allow your programming application to access STVP's low level DLLs.

With access to these STVP DLLs, you can program ST7 and STM8 microcontrollers using any supported programming hardware (ST7-STICK, ST socket boards, EPB, DVP3, EMU3, STice, ST-LINK, ST-TSLINK and Raisonance STX-RLINK) and programming methods (socket, in-circuit or in-situ programming).

Figure 1.   Programming application schematic
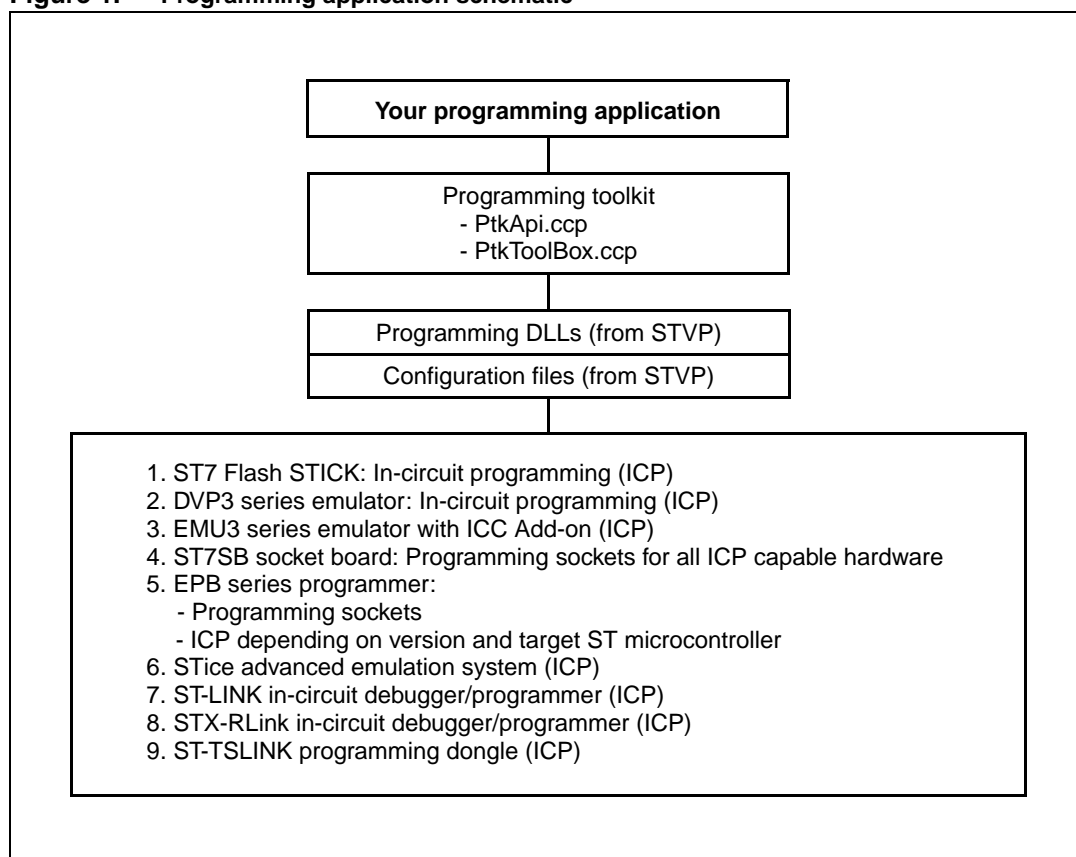
Note: *Creating a programming application with the STVP programming toolkit requires the installation of ST Visual Programmer version 1.9.1 or later.*

Contents

1 Overview

1.1 About the user manuals

This manual provides information and examples to help you develop your programming application with the programming toolkit. It describes:

● The functions supported by the STVP DLLs

● Using the functions provided by the programming toolkit

For information about STVP and supported ST programming hardware, you can refer to the following:

● ST Visual Programmer on-line help, information about STVP and supported hardware and programming methods

● *ST Visual Programmer Release Notes*, complete release information about the current release of STVP, including supported programming hardware and ST microcontrollers

For information about setup and connection information for supported programming hardware, you can refer to the following:

● *ST7-STICK User Manual*

● *ST7SB Socket Board User Manual*

● *ST7-EPB User Manual*

● *ST7-DVP3 Emulator User Manual*

● *ST7-EMU3 Emulator User Manual*

● *STice User Manual*

● *ST-LINK User Manual*

● *ST-TSLINK User Manual*

● *Raisonance STX-RLINK User Manual* (on the Raisonance web site)

For programming information that is specific to your ST microcontroller, refer to the relevant datasheet.

1.2 Getting assistance

For more information, application notes, FAQs and software updates on all the ST microcontroller families, check out the CD-ROM or our website: www.st.com

For assistance on all ST microcontroller subjects, or for help using your emulator, refer to the contact list provided in the *Chapter Appendix A: Product support*. We'll be glad to help you.

2 Programming toolkit contents

The programming toolkit is a set of C++ source and header files that you integrate into your application in order to use the programming functions supported by STVP's DLLs. In addition, the toolkit contains two samples that demonstrate the types of applications that can be created.

When you run the auto extracting zip, the programming toolkit files and this documentation are placed in the directory that you specified on your PC. These files and folders include:

Table 1.   Installed components: programming toolkit source and header files

src:	Contains the C++ source files that you include in your application in order to access the functions of the STVP DLLs.
PtkApi.cpp: PtkApi.h:	Source and header that enable the importation of functions from the application programming interfaces (API) of the STVP DLLs.
PtkToolBox.cpp PtkToolBox.cpp	Source and header for "Helper Functions" for tasks such as retrieving configuration files or accessing the "memory image." The Helper Functions should always be used to access the configuration information as the format and content of these databases may change without notification when STVP is upgraded. Even if the format of the database changes, the prototypes for the Helper Functions will remain the same.

Note: *The source files provided in the programming toolkit are in C++ format (.cpp), however they can be compiled as C source files. Developing your application using another programming language does not necessarily pose a compatibility problem with the STVP DLLs, however these developers will have to rewrite the application programming interface (PtkApi.cpp).*

Programming applications created using the programming toolkit rely on STVP's DLLs and configuration files (see the cover page). The programming toolkit does not contain these files; they are installed with STVP. These files must then be placed in the same directory as your programming application. More information about these DLLs is provided in the next chapter, *Understanding the STVP DLLs*.

Note: *Both the **STVP programming toolkit** and the latest version of **ST Visual Programmer** can be downloaded for free from **www.st.com/mcu**.*

Table 2.   Source files and executables for two sample applications:

Folder/File	Description
Sample:	The samples, APISample.exe and MFCSample.exe, are C applications developed with Visual C++ 6.0. They should be executed from the directory where you have installed STVP version 1.9.1 or later.
ApiSample:	Contains the C++ source and header files for a console application that is intended to illustrate: • How to set the hardware and MCU configuration: ST7FLITE29, ST7-STICK, LPT1, "ICP OPT Enable" protocol • How to load binary files in "PROGRAM MEMORY" and in "OPTION BYTE" areas • How to program and verify the "PROGRAM MEMORY" • How to program and verify the "OPTION BYTE" This sample can easily be rebuilt in an environment other than Visual C++ 6.0.
Readme.txt	Provides details about the use of the ApiSample application.
Distrib:	This subdirectory contains the executable and .hex files for the ApiSample. The files include:
ApiSample.exe:	The executable for the application programming interface. This executable should be executed from the directory where you have installed STVP.
option.hex:	Dummy application code that the sample application programs to the option byte area of the ST microcontroller's memory.
progmem.hex:	Dummy application code that the sample application programs to the ST microcontroller's Program Memory area.
MFCSample:	Contains the C++ source and header files for an application that uses the Microsoft Foundation Classes. In addition to the features demonstrated in ApiSample, MFCSample specifically illustrates the use of the helper functions to access and configure data.
Readme.txt:	Provides details about the use of the MFCSample application.
Distrib:	This subdirectory contains the MFCSample executable file.
MFCSample.exe:	The executable for the application programming interface. This executable should be executed from the directory where you have installed STVP.
res:	Contains the resource files for the graphical interface, which were used to build the sample application.

3 Understanding the STVP DLLs

Because your programming application must use the STVP DLLs, you should first understand their roles. The software elements that your application accesses and their relationships are shown in *Figure 2*. These elements include:

Table 3. STVP DLLs and configuration files

Configuration files	
DBCFILE.CNF	Contain a database of information on all STVP supported devices (memory mappings, option bytes, protocols) and programming hardware configurations (supported devices, protocols, communication ports). Typically, these files evolve with new releases of STVP. However, if you use the programming toolkit, these evolutions should not pose a problem for your application.
TOOLS.CNF	
DLLs	
DBCA60.DLL	**Database configuration access**. Used by HAPL, EPRCORE60 and programming toolkit to get lists of devices, programming hardware and other hardware and device information from the configuration files.
EPRCORE60.DLL	**Epromer core**. Keeps track of the device description and creates the programming interface between the GUI and the appropriate HAPL.
FILE60.DLL	A component DLL of the EPRCORE60 layer. It ensures loading and saving of the device binary files. Supported formats include Motorola (.S19) and Intel (.HEX).
HAPL.DLL	**Hardware programming level**. Hardware specific DLLs that are in charge of programming a specific type of device using a specific type of hardware. The EPRCORE60 loads the appropriate HAPL for the hardware and device configuration.
LEF60.DLL	**Logging / Error features**. Ensures the propagation of errors between the different layers. It may also log execution traces.

Figure 2. STVP DLLs and configuration files
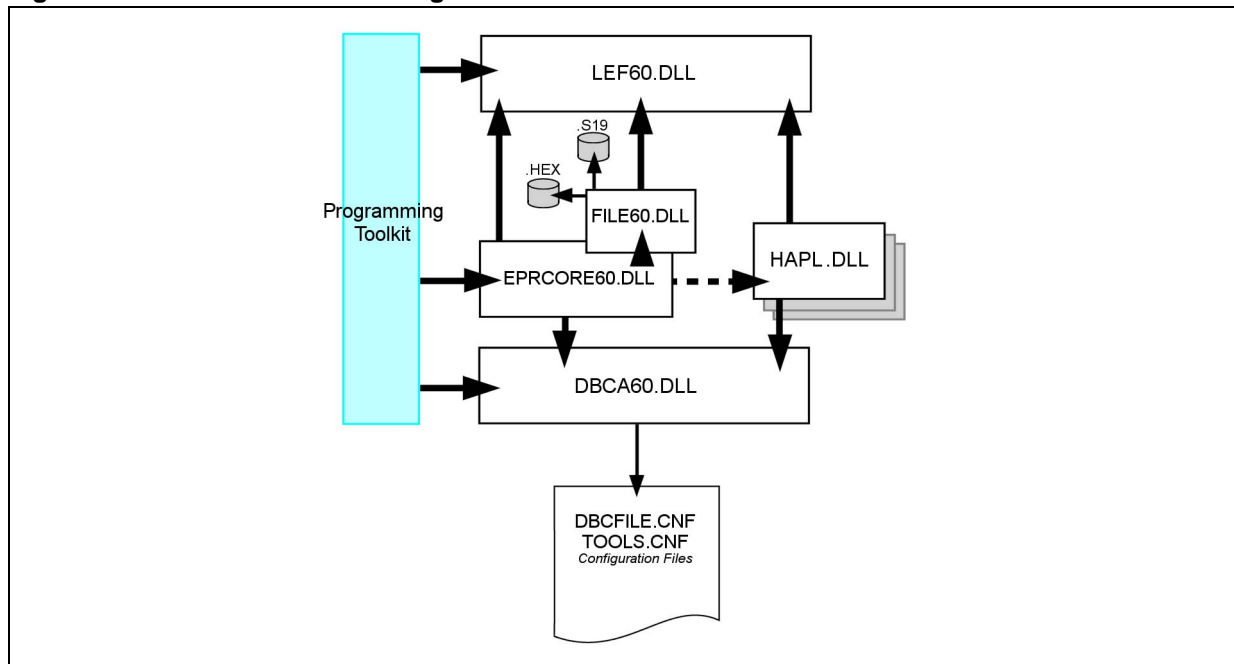
4   Designing a custom programming application

For the purposes of understanding the different functions supported by the STVP DLLs, the programming process can be divided into four categories of functions:

● *Setting up the DLL environment*
● *Configuring the device and the programming hardware*
● *Accessing the memory image*
● *Connecting to the device*

The following sections provide coded examples to show how an application might use the different functions in each category to accomplish specific programming tasks.

Before getting started, it is also important to recognize that your application can be **dynamic**, offering as much control of hardware configuration and options as STVP, or **static**, with the hardware configuration hard-coded into the programming application. The programming toolkit provides **Helper Functions** that allow you to return lists of devices, hardware and protocols at run time so that they can be used by your application (refer to *Programming toolkit helper functions*). However, for simplicity, the examples provided in the following section are for a static interface with the configuration hard-coded into the application.

4.1   Setting up the DLL environment

The STVP DLLs described in *Section 3* must be initialized by your application and the callback functions that allow it to manage errors, messages and process execution must also be initialized.

The following is a standard initialization that you might use in your application.

long CALLBACK AppendMessageText(const char* szMsg)

```
{
.....
}
long CALLBACK AppendErrorText( const char* szMsg)
{
.....
}
long CALLBACK AppendWarningText( const char* szMsg)
{
.....
}
long CALLBACK UpdateProgress( int percentage* szMsg)
{
.....
}
```

Note: *The CALLBACK notation used here is standard when using Microsoft Visual Studio C++. However, for other development environments, you may have to use a different notation.*

```
InitDLLCallBack()
{
```

```
      LSetErrorCallBack( AppendErrorText);
      LSetWarningCallBack( AppendWarningText);
      LSetMessageCallBack( AppendMessageText);
      LSetProgressCallBack( UpdateProgress);
}
main() {
// Load DLLs and Initialize the address of API
// functions
if (LoadDlls("C:\my_installation\STVP\")==1) {
   InitDLLCallBack();
   .....
   // Log everything
   LOpenLog( "Activity.log");
   .....
   SetProgrammingConfiguration();
   DoesProgrammingJob();
   .....
   LCloseLog();
   Clean();
   }
}
```

4.2 Configuring the device and the programming hardware

The STVP DLLs support numerous hardware configurations and target devices. Correct identification of the programming hardware is important in establishing the necessary hardware connections, as well as programming and verification of the target device. Once the configuration is set, the correct HAPL.DLL is loaded to manage communication with the device.

Based on the device selected for programming, the programming software also creates buffer area in your PCs RAM for the data that will be programmed to the device. This buffer is called the **Memory Image**. It is also used as a model to check against when verifying the programming of the device.

The routine below is initiated by the *SetProgrammingConfiguration()* function. This is an example of a static interface where the following configuration is hard-coded into the application:

> Hardware: ST7MDT10-EPB
>
> Protocol: ICP OPT Enable
>
> Device: ST7FLITE09
>
> Port: LPT1

Note: *This example is incomplete as some parts of the code are implementation-driven.*

```
BOOL SetProgrammingConfiguration()
{
// Select the hardware in this case ST7MDT10-EPB with
// "ICP OPT Enable"
// protocol, FALSE means that we are not using
// simulation mode
if (ESelectHard("St7MDT10-EPB", "ICP OPT Enable" ,FALSE)==0)
```

```
    {
    return false;
    }
// Prevent from opening a warning dialog box when
// Option byte protection bit is to be programmed.
// By default, the Protection warning dialog box is
// opened
if (ESetPreferences (PROTECTION_WARNING, FALSE)==0}
    {
    return false;
    }
// Select the device : ST7FLITE09 device
if (ESelectDevice("ST7FLITE09")==0))
    {
    return false;
    }
if (ESelectPort ("LPT1")==0))
    {
    return false;
    }
// Set the protection for RC Calibration on program
// memory
if (ESetProtection("Protect RC Calibration values on Program
Memory", TRUE)==0)
    {
    return false;
    }
    return true;
}
```

4.3 Accessing the memory image

Once the programming hardware, the device and the port have been specified, the EPRCORE60.DLL allocates memory structures and buffers for each memory area supported by the device (PROGRAM, DATA, OPTION). The memory buffer internally allocated by EPRCORE60 is called the **Memory Image**. A unique internal identifier is given to each area. This identifier should be used later in the API functions whenever you need to identify a specific memory area.

With the Memory Image established, the programing application can now access this buffer and load the binary file (.HEX or .S19) that you wish to program to your device, as well as writing any information that is specific to the programming of each device (ex. a product serial number).

Caution: If your software reads your ST microcontroller's memory, the data from the device overwrites any data that is already stored in the Memory Image.

4.3.1 Loading a file

In the following example the specified binary file (MyAppli.S19), which is to be programmed to the target device, is retrieved and saved to the Memory Image. This file is located and saved by the FILE60 component of the EPRCORE60.

```
// Search the PROGRAM MEMORY ID
unsigned long iAreaId
if (EGetId("PROGRAM_MEMORY", &iAreaId) == 1)
   {
   // Load the file MyAppl.S19 in the program memory area
   iReturn = ELoadFile("MyAppl.S19",iAreaId);
   }
```

4.3.2   Writing in the memory image

You can also write punctually to specific memory locations as shown in this example. In the sample provided the data is an integer that is incremented each time a device is programmed. For example, this value could be a product serial number or other identification.

```
.....
// Write 1,2,3,4 in memory image at location 0x2000
for (i=1;i<=4;i++)
   {
   SetByteInImageMemory(ST7FLITE09, "PROGRAM MEMORY",0x2000+i- 1,i);
   }
.....
```

4.4   Connecting to the device

The following examples use the functions that interact with the device that is to be programmed. At this stage your programming application is writing to the device's memory, the data which has been stored in the Memory Image. This section provides examples specifically for blank checking, programming, verifying and reading a device.

4.4.1   Blank checking the device

Blank checking the device's memory allows you to confirm that the device has not already been programmed. Some devices do not support blank check. To avoid an error when not supported, you can use the *IsBlankCheckAvailable* function as in illustrated in the example below.

```
// Search the PROGRAM MEMORY ID
unsigned long iAreaId
// IsBlankCheckAvailable will prevent from executing
// the blank check, when the MCU does not support blank
// check which is the case for  the current
// configuration (ST7FLITE09)
if (IsBlankCheckAvailable("ST7FLITE09","PROGRAM MEMORY")==true)
   {
   if (EGetId("PROGRAM_MEMORY", &iAreaId) == 1)
      {
      // Process all the PROGRAM MEMORY
      EBlankAll(iAreaId);
      // Blank check between 0x2000 and 0x3000
      EBlankArea( iAreaId, 0x2000L, 0x3000L);
      }
   }
```

4.4.2   Programming the device

The programming routine writes the contents of the Memory Image to the specified areas of the device's memory.

```
// Search the PROGRAM MEMORY ID
unsigned long iAreaId
if (EGetId("PROGRAM_MEMORY", &iAreaId) == 1)
   {
   // Process all the PROGRAM MEMORY
   EProgAll(iAreaId);
   // Program only between 0x2000 and 0x3000
   EProgArea(iAreaId, 0x2000L, 0x3000L);
   }
```

4.4.3   Verifying the programming of the device

The verification routine compares the contents of the programmed device memory with the data as you intended to program it. This is the data stored in the Memory Image.

Caution: It is very important not to read the device's memory prior to verification. When reading the device, the contents of the device's memory overwrite the contents of the Memory Image, which are used for verification.

```
// Search the PROGRAM MEMORY ID
unsigned long iAreaId
if (EGetId("PROGRAM_MEMORY", &iAreaId) == 1)
   {
   // Process all the PROGRAM MEMORY
   EVerifyAll(iAreaId);
   // Read between 0x2000 and 0x3000
   EVerifyArea(iAreaId, 0x2000L, 0x3000L);
   }
```

4.4.4   Reading the device

A read routine allows you to read the contents of the target device's memory.

```
// Search the PROGRAM MEMORY ID
unsigned long iAreaId
if (EGetId("PROGRAM_MEMORY", &iAreaId) == 1)
   {
   // Process all the PROGRAM MEMORY
   EReadAll(iAreaId);
   // Read between 0x2000 and 0x3000
   EReadArea(iAreaId, 0x2000L, 0x3000L);
   }
```

Caution: Do not to read the device prior to verification. When reading the device, the contents of the device's memory are stored in the Memory Image, replacing the data that would have been used for verification.

5   DLL supported functions

The following sections provide a detailed description of the programming toolkit's DLL functions, including those used in the preceding examples. The programming toolkit's DLL functions are intended to ease access to the STVP programming DLLs.

As a general rule, you can determine which DLL a function calls, by looking at the first letter of the function. For example:

Table 4.   Examples of determining which DLL a function calls

Functions starting with	Call	For example
C	DBCA60.DLL	CSetWorkingDir, CGetLastError
E	EPRCORE60.DLL	ESelectDevice, EProgAll
L	LEF60.DLL	LOpenLog, LDisplayError

To help you understand their use, we've divided the DLL functions into four main categories:

● *DLL environment functions*
● *Hardware and device configuration functions*
● *Image area access functions*
● *Device connection functions*

5.1   DLL environment functions

These functions may be called by your programming application to handle errors, control the process execution, or log the result of basic device actions.

CGetLastError

Returns the last string error occurred after a query to the configuration files

Prototype: char* CGetLastError()

Return: The error string

Clean

Unloads STVP and frees all the resources allocated by the *LoadDlls* function.

Prototype: void Clean ()

Parameters: None

CSetWorkingDir

Sets the directory where configuration files will be searched for. By default, the configuration files will be searched for in the directory of the calling application.

Prototype: int CSetWorkingDir(const char* szPath)

Parameters: szPath: requested path

Return: 0 if error, 1 if success

LCloseLog

Closes the currently opened log File

Prototype: int LCloseLog()

Return: 0 if error, 1 if success

LDisplayError

This function is used by the STVP DLLs to display errors. It also logs the error if a log session has been opened. LDisplayError calls the function that has been initialized by LSetErrorCallBack.

If your application has defined a callback for LDisplayError, this function should be given priority over the internal function, because using it will exercise the logging mechanism.

Prototype: int LDisplayError(const char* szMessage)

Parameters: szMessage: error message text.

Return: 0 if error, 1 if success

LDisplayMessage

This function is used by the STVP DLLs to display messages. It also logs the message if a log session has been opened. In fact LDisplayMessage calls the function that has been initialized by LSetMessageCallBack.

If your application has defined a callback for LDisplayMessage, this function should be given priority over the internal function, because using it will exercise the logging mechanism.

Prototype: int LDisplayMessge(const char* szMessage)

Parameters: szMessage: message text

Return: 0 if error, 1 if success

LDisplayWarning

This function is used by the STVP DLLs to display warnings. It also logs the message if a log session has been opened. LDisplayWarning calls the function that has been initialized by LSetWarningCallBack.

If your application has defined a callback for LDisplayWarning, this function should be given priority over the internal function, because using it will exercise the logging mechanism.

Prototype: int LDisplayWarning(const char* szMessage)

Parameters: szMessage: warning message text

Return: 0 if error, 1 if success

LoadDlls

Loads the STVP DLLs and retrieves the API addresses. It is not exactly an API function but it should be called prior to any API function otherwise the API function address will not be initialized

Prototype: int BOOL LoadDlls(const char* szDLLPath)

Parameters: szDLLPath: The path to access the STVP Programming DLLs

Return:

-1: if error (at least one DLL not loaded or at least one API function not found)

1: if success

LOpenLog

Opens a log file.

If your application has defined callbacks for LDisplayError, LDisplayWarning and LDisplayMessage, these functions should be given priority over internal functions, because using them will exercise the logging mechanism.

Prototype: int LOpenLog(const char* szFileName)

Parameters: FileName: name of the log file

Return: 0 if error, 1 if success

LSetErrorCallBack

Sets up the error callback function. This function is called by the STVP programming DLLs each time an error occurs during an API function call. The error routine handler should be used to display errors issued by the programming DLLs.

It is not necessary for your application to execute this function, if the error handler has not been set up, nothing is done with the error message and it is lost.

Prototype: int LSetErrorCallBack(long *fonct(const char* szMessage))

Parameters: fonct: Points to the display error handle. The prototype for this function is:

```
long  CALLBACK fonct(const char* szMessage)
```

Return: 0 if error, 1 if success

LSetLogOptions

Configures logging options so only certain kinds of information (Errors, warnings, messages) are stored:

Before using LSetLogOptions, open the logging file name first — *LOpenLog*.

Prototype: int LSetLogOptions(USHORT Options, OPERATOR ope)

Parameters: Options: One of the following state words defining the messages to log —

LOG_MSG: 1
LOG_WARN: 2
LOG_ERR: 4
LOG_ALL: LOG_MSG | LOG_WARN | LOG_ERR

ope: Enum values 'OR' or 'AND' set and remove an option respectively.

typedef enum {OR, AND} OPERATOR

For example:

```
.....
LOpenLog( "Activity.log");
```

```
if (!LSetLogOptions( LOG_ALL, OR)) {
// Log everything
.....
}
.....
// Do not log warnings anymore.
SetLogOption(~LOG_WARN,AND);
.....
```

Return: 0 if error, 1 if success

LSetMessageCallBack

Sets up the Message callback function. This function is called by the STVP programming DLLs each time a message occurs during an API function call. Typically the routine handler should be used to display the messages issued by the programming DLLs.

It is not necessary for your application to execute that function, but if the message handler has not been set up the message is lost.

Prototype: int LSetMessage CallBack(long *fonct(const char* szMessage))

Parameters: fonct: Points to the function handler. The prototype for this function is:
```
long CALLBACK fonct(const char* szMessage)
```

Return: 0 if error, 1 if success

LSetProgressCallBack

This function may used by your application to set up a progress callback function. This function is invoked by the STVP programming DLLs to generate an action progress.

Prototype: int LSetProgress CallBack(long *fonctInt Percentage))

Parameters: fonct: Points to the "in progress" function handler: the prototype for that function must be:
```
long CALLBACK fonct(int iPercent)
```

Return: 0 if error, 1 if success

LSetWarningCallBack

Sets up the warning callback function. This function is called by the STVP programming DLLs each time a warning occurs during an API function call. The routine handler should be used to display the warnings issued by the programming DLLs

It is not necessary for your user application to execute that function, if the warning handler has not been set up, nothing is done with the warning and it is lost.

Prototype: int LSetWarningCallBack(long *fonct(const char* szMessage))

Parameters: fonct: Points to the function handler. The prototype for this function is:
```
long CALLBACK fonct(const char* szMessage)
```

Return: 0 if error, 1 if success

LTraceLog

Writes a message in the currently opened log file

Prototype: int LTraceLog(const char* Message)

Parameters: Message: The text of the logged message

Return: 0 if error, 1 if success

5.2 Hardware and device configuration functions

These functions allow you to select and configure both the programming hardware (EPB, ST7-STICK, DVP3, EMU3, STice, ST-LINK, ST-TSLINK and Raisonance STX-RLINK), device, protocol and communication port that will be used during the programming session. The list of supported hardware, ports, protocols and devices are the same as those supported by STVP. These are displayed in the **Configure ST Visual Programmer** dialog box. You can also retrieve this information at run time using the *Programming toolkit helper functions*.

ESelectDevice

Selects the device to be programmed.

Prototype: int ESelectDevice(const char* szDevice)

Parameters: szDevice: device name

Return: 0 if error, 1 if success

ESelectHard

Select the programming hardware (EPB, DVP3, EMU3, ST7-STICK, STice, ST-LINK, ST-TSLINK or Raisonance STX-RLINK) and the protocol to be used.

Note: *The list of available protocols on the target hardware and device. You will find this information in STVP's* **Configure ST Visual Programmer** *dialog box, or you can use the function* GetProtocolList *provided by the programming toolkit.*

Prototype: int ESelectHard(const char* szCard, const char* szProtocol,BOOL bDemo)

Parameters:

   szCard: hardware programming board

   szProtocol: protocol name

   bDemo: TRUE for device simulation, meaning that the programming software does not connect to the programming hardware.

Return: 0 if error, 1 if success

ESelectPort

Selects the communication port.

Prototype: int ESelectPort(const char* szPort)

Parameters: szPort: The port name (LPT1, USB, etc.)

Return: 0 if error, 1 if success

ESetPreferences

Enables or disables the preferences. Only "Protection Warnings" can be disabled/enabled for the moment. Disabling prevents a dialog box from being opened when the protection bit in the Option byte is to be programmed.

Note: *This function must be called after the* ESelectHard *function call and before each* ESelectDevice *function call.*

Prototype: int ESetPreferences(int iPreference, BOOL bState)

Parameters:

Preference: Preference for example: PROTECTION_WARNING=0x01)

bState: TRUE or FALSE

Return: 0 if error, 1 if success

ESetProtection

On some MCU, it enables or disables protection on specific addresses for instance the RC Calibration on ST7FLITE09. When a protection is set, the relevant addresses are not programmed

Note: *Protection mode is expressed as a string. The list of protection modes can be found either in STVP's* **Configuration** *dialog box, or at run time by using the GetProtectionList function from the* **Programming Toolkit***.*

Prototype: int ESetProtection(const char* szProtectMode, BOOL bState)

Parameters:

szProtectMode: The literal string for the protection mode.

bState: TRUE to enable or FALSE to disabled the protection.

Return: 0 if error, 1 if success

5.3 Image area access functions

The following functions allow you to retrieve and store information in the buffer called the Memory Image that the programming software creates on your PC.

ECheckSum

Returns the checksum from the last file loaded and the current memory checksum

Prototype: int ECheckSum(DWORD dwAreaId, long *FileCheckSum, long *MemoryCheckSum)

Parameters:

dwAreaId: Area identifier

FileCheckSum: Last file checksum

MemoryCheckSum: Current memory checksum

Return: 0 if error, 1 if success

EGetId

Returns the unique identifier for the specified area.

Prototype: int EGetId(const char* szName, DWORD* pdwId)

Parameters:

szName: Literal name for the programming area (For example, PROGRAM MEMORY, or OPTION BYTE).

pdwId: Pointer with the area identifier

Return: 0 if error, 1 if success

EGetImagePtr

Returns the pointer the size of the specified area in the Memory Image.

Prototype: char far *EGetImagePtr(DWORD dwAreaId, long *MemSize)

Parameters:

dwAreaId: Area identifier

MemSize: Memory size

Note: *Be careful when using a pointer to the memory image buffer, you should use the physical address for the area that you want to access as indexed in the buffer. The programming toolkit's GetByteInImageMemory or SetByteInImageMemory helper functions should be used instead of read or write in the memory image functions.*

Return: The memory image pointer is NULL if non existent memory

For example:

```
// Write 0x5A in PROGRAM MEMORY memory image at adress 0x8000
char* pImgMemory;
unsigned long dwAreaId
long lMemSize;
if (EGetId("PROGRAM MEMORY",&dwAreaID)==1)
   {
   pImgMemory = EGetImagePtr(dwAreaID,&lMemSize)
   if (pImgMemory)
      {
      pImgMemory[0x8000] = 0x5A;
      }
   }
```

ELoadFile

Loads a binary file in the memory image. Supported binary formats are Motorola .S19 and Intel .HEX

Prototype: int ELoadFile(const char* FileName, DWORD dwAreaId)

Parameters:

FileName: The name of the binary file

dwAreaId: Area identifier, see *EGetId*.

Return: 0 if error, 1 if success

ESaveFile

Saves the memory image in a binary file

Note: *The binary format used to save the file depends on the extension of the FileName parameter. The .S19 extension will generate a Motorola S19 file, .HEX extension generates an Intel HEX binary file*

Prototype: int ESaveFile(const char* FileName, DWORD dwAreaId)

Parameters:

FileName: The binary file name

dwAreaId: Area identifier

Return: 0 if error, 1 if success

5.4   Device connection functions

This section describes all the functions that physically interact with the device that is to be programmed: blank check, program, verify and read.

● Blank check ensures that the device memory area is programmed with the default value.

● Programming the device programs the device's memory with the values currently stored in the Memory Image.

● Verify compares the value for each address of a specified area in the Memory Image, with that stored in the device's memory.

● Reading the device fills the Memory Image for the specified area with values read from the device's memory.

EBlankAll

Blank checks the entire target memory area. Works only on FLASH and EPROM memory.

Prototype: int EBlankAll(DWORD dwAreaId)

Parameters: wAreaId: Area identifier

Return: 0 if error, 1 if success

EBlankArea

Blank checks a part of the device's memory.

Prototype: int EBlankArea(DWORD dwAreaId,long FirstAddr,long LastAddr)

Parameters:

dwAreaId: Area identifier

FirstAddr: 1st device memory address

LastAddr: Last device memory address

Return: 0 if error, 1 if success

EEraseAll

Erases all the sectors of FLASH memory.

Prototype: int EEraseAll(DWORD dwAreaId)

Parameters: dwAreaId: Area identifier.

Return: 0 if error, 1 if success

EEraseArea

Erases a target FLASH memory sector. *FirstAddr* and *LastAddr* must map the sector to be erased exactly.

Prototype: int EEraseArea(DWORD dwAreaId,long FirstAddr,long LastAddr)

Parameters:

dwAreaId: Area identifier
FirstAddr: 1st device memory address
LastAddr: Last device memory address

Return: 0 if error, 1 if success

EProgAll

Programs the entire device memory.

Prototype: int EProgAll(DWORD dwAreaId)

Parameters: dwAreaId: Area identifier

Return: 0 if error, 1 if success

EProgArea

Programs a part of the device's memory. Do not use this function to program the OPTION BYTE. The OPTION BYTE must be programmed all at once using the *EProgAll* function and not bit-by-bit.

Prototype: int EProgArea(DWORD dwAreaId, long FirstAddr,long LastAddr)

Parameters:

dwAreaId: Area identifier
FirstAddr: 1st device memory address
LastAddr: Last device memory address

Return: 0 if error, 1 if success

EReadAll

Reads the entire device memory area.

Prototype: int EReadAll(DWORD dwAreaId)

Parameters: dwAreaId: Area identifier

Return: 0 if error, 1 if success

EReadArea

Reads a part of the device's memory area. Do not use this function to read the OPTION BYTE. The full OPTION BYTE must be read at once using the *EAllRead* function and not bit-by-bit.

Prototype: int EReadArea(DWORD dwAreaId, long FirstAddr, long LastAddr)

Parameters:

dwAreaId: Area identifier

FirstAddr: 1st device memory address

LastAddr: Last device memory address

Return: 0 if error, 1 if success

EVerifyAll

Verifies the entire memory area of the programmed device.

Prototype: int EVerifyAll(DWORD dwAreaId)

Parameters: dwAreaId: area identifier

Return: 0 if error, 1 if success

EVerifyArea

Verifies a part of the target memory area. Do not use this function to verify the OPTION BYTE. The full OPTION BYTE must be verified at once using the *EVerifyAll* function and not bit-by-bit.

Prototype: int EVerifyArea(DWORD dwAreaId, long FirstAddr,long LastAddr)

Parameters:

dwAreaId: area identifier

FirstAddr: 1st device memory address

LastAddr: Last device memory address

Return: 0 if error, 1 if success

6 Programming toolkit helper functions

This section describes the functions that extract the most useful requests to the configuration file databases and a few functions that allow the application to write to and read the Memory Image. These functions are located in the PtkToolBox.cpp C source file.

Note: *The functions below should be used to access to the configuration information as the database file format may change without notification. Even if the format of the database changes, the prototype for the Helper Functions will remain the same.*

FillImageMemory

Fills the memory image with the blank value.

Prototype: int FillImageMemory(const char* szDeviceName, const char* szAreaName)

Parameters:

    szDeviceName: device name
    szAreaName: name of the area

Return: 0 if error, 1 if success

GetAreaList

Fills the output buffer with the list of memory areas (PROGRAM, DATA, OPTION...) that are supported by a particular device. In the output buffer, the items in the list are separated by '\n'. When the function returns an error, the error is due to a database access problem. Use the CGetLastError function to retrieve the error message.

Prototype: int GetAreaList(const char*szDeviceName, char* szListBuffer, int iBufferSize)

Parameters:

    szDeviceName: device name
    szListBuffer: output buffer that contains the list of memory areas
    iBufferSize: size of the output buffer

Return: 0 if error, 1 if success

GetAreaMapping

Returns the address range (first and last addresses) of a given memory area. If the area mapping is split in several sectors, FirstAddr contains the lower address of the first section and LastAddr contains the higher address of the last section.

Prototype: int GetAreaMapping(const char* szDevice,const char* szAreaName, unsigned long *FirstAddr, unsigned long LastAddress)

Parameters:

    szDeviceName: device name
    szAreaName: name of the area
    FirstAddr: output parameter that contains the first area address
    LastAddr: output parameter that contains the last area address

Return: 0 if error, 1 if success

GetByteFromImageMemory

Reads a byte at a given address in the memory image.

Prototype: int GetByteFromImageMemory(const char* szDeviceName, const char* szAreaName, unsigned long dwAddr, unsigned char* uValue)

Parameters:

szDeviceName: device name

szAreaName: name of the area

dwAddr: memory image address to be written

uValue: byte value to write

Return: 0 if error, 1 if success

GetDeviceList

Fills the output buffer with the list of devices supported by a particular protocol for a particular programming hardware.

Note: *In the Output buffer, the items in the list are separated with '\n'*

When the function returns an error, the error is due to a database access problem. Use the CGetLastError function to retrieve the error message.

Prototype: int GetDeviceList(const char* szHardName, const char* szProtocolName, char* szListBuffer, int iBufferSize)

Parameters:

szHardName: board name

szProtocolName: name of the protocol to be used

szListBuffer: output buffer that contains the list of devices matching with the board and the protocol

iBufferSize: size of the output buffer

Return: 0 if error, 1 if success

GetHardwareList

Fills a buffer with the list of programming hardware currently supported by the configuration database. In the Output buffer, the items in the list are separated with '\n' .
When the function returns an error, the error is due to a Data base access problem, use the CGetLastError function to get the error message

Prototype: int GetHardwareList(char* szListBuffer, int iBufferSize)

Parameters:

szListBuffer: output buffer that contains the list of programming hardware

iBufferSize: size of the output buffer

Return: 0 if error, 1 if success

GetNextMapSector

Iterates on all sectors from the area mapping. When initiating the iteration, the GetNextMapSector function should be called with szDevice and szAreaName initialized.

During the iteration, GetNextMapSector should be called with a NULL value for szDevice and szAreaName.

Prototype: int GetNextMapSector((const char* szDevice,const char* szAreaName, unsigned long *FirstAddr, unsigned long LastAddress)

Parameters:

szDeviceName: device name

szAreaName: name of the area

FirstAddr: output parameter that contains the first sector address

LastAddr: output parameter that contains the last sector address

Return:

0 if error

1 if there is at least one more sector

-1 if the end of the mapping has been reached

For example

```
unsigned long FirstAreaAddr,lLastAreaAddr;
any = GetNextMapSector("ST7FLCD1","PROGRAM MEMORY",
&FirstAddr,LastAreaAddr);
while (any == 1)
   {
   any = GetNextMapSector(NULL,NULL,&FirstAreaAddr,
&lLastAreaAddr);
   }
// Reach end of map
```

GetPortList

Fills the output buffer with the list of communication ports that may be used to connect with a particular board. In the Output buffer, the items in the list are separated by '\n'.

When the function returns an error, the error is due to a database access problem. Use the CGetLastError function to retrieve the error message.

Prototype: int GetPortList(const char* szHardName, char* szListBuffer, int BufferSize)

Parameters:

szHardName: name of the programming hardware

szListBuffer: output buffer that contains the list of ports supported by the programming hardware

iBufferSize: size of the output buffer

Return: 0 if error, 1 if success

GetProtectionList

Fills the output buffer with the list of protections supported by a given device.

Note: *Protection is an optional state that allows user to protect some reserved addresses, for instance the RC Calibration bytes for some ST7LITE microcontrollers. When protection is set, the relevant addresses are not programmed (see ESetProtection).*

In the Output buffer, the items in the list are separated by '\n'.

When the function returns an error, the error is due to a database access problem. Use the CGetLastError function to retrieve the error message.

Prototype: int GetProtectionList(const char*szDeviceName, char* szListBuffer, int iBufferSize)

Parameters:

szDeviceName: device name

szListBuffer: output buffer that contains the list of protection options supported by the device

iBufferSize: size of the output buffer

Return: 0 if error, 1 if success

GetProtocolList

Fills the output buffer with the list of protocols supported by a particular board. In the output buffer, the items in the list are separated by '\n'. When the function returns an error, the error is due to a database access problem. Use the CGetLastError function to retrieve the error message.

Prototype: int GetProtocolList(const char* szHardName,char* szListBuffer, int iBufferSize)

Parameters:

szHardName: board name

szListBuffer: output buffer that contains the list of protocols supported by the programming board

BufferSize: size of the output buffer

Return: 0 if error, 1 if success

IsBlankCheckAvailable

Indicates whether a given area may be blank checked. Blank check can be done only on Flash and EPROM areas

Prototype: int IsBlankCheckAvailable(const char* szDeviceName, const char* szAreaName)

Parameters:

szDeviceName: device name

szAreaName: name of the area

Return:

0: blank check is not allowed on the area

1: if the area may be blank checked

IsErasableArea

Indicates whether a given area may be erased. Only Flash areas are erasable.

Prototype: int IsErasableArea(const char* szDeviceName, const char* szAreaName)

Parameters:

szDeviceName: device name

szAreaName: name of the area

Return:

0: area is not erasable

1: area may be erased

SetByteInImageMemory

Writes a byte in the memory image at a given address.

Prototype: int SetByteInImageMemory(const char* szDeviceName, const char* szAreaName, unsigned long dwAddr, unsigned char uValue)

Parameters:

szDeviceName: device name

szAreaName: name of the area

dwAddr: memory image address to be written

uValue: byte value to write

Return: 0 if error, 1 if success

Appendix A   Product support

If you experience any problems with this product, or if you need spare parts or repairs, contact the distributor or the STMicroelectronics sales office where you purchased the product. Phone numbers for major sales regions are provided on the *www.st.com* web site.

From the *www.st.com* site, select **Products > Microcontrollers** to obtain a complete online selection guide, as well as documentation, software downloads and user discussion groups to help you answer questions and stay up to date with our latest product developments.

Software updates

All our latest software and related documentation are available for download from the STMicroelectronics internet site, *www.st.com*.

If you are using software from a third-party tool provider, please refer to the third-party for software product support and downloads.

7 Revision history

Table 5.   Document revision history

Date	Revision	Changes
01-Apr-2004	1	Initial release.
15-Jul-2005	2	– Added ST7SB Socket Board to supported hardware in Introduction – Updated Figure 1 – showing supported programming hardware – Updated About the user manuals – list of supporting documentation – Updated internet address in Getting assistance – Updated Table 1 – installed components
06-Apr-2009	3	Revalidation – Corrected revision numbers – Updated to new template
21-Jan-2010	4	– Updated to new template – Updated to support STM8, STice, ST-LINK, ST-TSLINK and Raisonance STX-RLINK

Please Read Carefully: