MA000-095-00-00
Doc. ver.: 1.13

EDE

Embedded Development Environment Manual

TASKING
Embedded software development from Altium.

A publication of

Altium BV

Documentation Department

http://www.tasking.com
http://wwww.altium.com

The information in this document has been carefully reviewed and is believed to be accurate and reliable. However, Altium assumes no liabilities for inaccuracies in this document. Furthermore, the delivery of this information does not convey to the recipient any license to use or copy the software or documentation, except as provided in an executed license agreement covering the software and documentation.

Altium reserves the right to change specifications embodied in this document without prior notice.

CONTENTS

TABLE OF CONTENTS

TASKING

CONTENTS

OVERVIEW 1-1

GETTING STARTED WITH EDE 2-1

USE EDE WITH CODEWRIGHT 3-1

EDITOR 4-1

PYTHON A-1

INDEX

MANUAL PURPOSE AND STRUCTURE

PURPOSE

This manual is aimed at users of the TASKING Embedded Development Environment (EDE). It assumes you are familiar with the Windows 95/98/NT/2000 graphical user interface.

MANUAL STRUCTURE

Related Publications
Conventions Used In This Manual

1. Overview
   Highlights specific EDE features and capabilities, and shows the EDE program development flow.

2. Getting Started with EDE
   Gives you information how to start and use EDE.

3. Using EDE with CodeWright
   Describes how to use EDE with an existing version of CodeWright.

4. Editor
   Describes the EDE editor and gives some help information and practical tips.

A. Python
   Describes how to extend EDE to use Python scripts.

INDEX

RELATED PUBLICATIONS

- CODE COMPLETE
  A Practical Handbook of Software Construction
  [Steve McConnel, Microsoft Press]
- On–line CodeWright Documentation [Starbase/TASKING]
- PVCS Documentation [Intersolv]
- SourceSafe Documentation
- TLIB Documentation
- RCS Documentation
- C and C++ Cross–Compiler User's Manuals [TASKING]
- Cross–Assembler User's Manuals [TASKING]
- C Compiler, Assembler, Linker Reference Manuals [TASKING]
- CrossView Pro Debugger User's Manuals [TASKING]

CONVENTIONS USED IN THIS MANUAL

The notation used to describe the format of call lines is given below:

{} Items shown inside curly braces enclose a list from which you must choose an item.

[ ] Items shown inside square brackets enclose items that are optional.

| The vertical bar separates items in a list. It can be read as OR.

italics Items shown in italic letters mean that you have to substitute the item. If italic items are inside square brackets, they are optional. For example:

filename

means: type the name of your file in place of the word *filename*.

... An ellipsis indicates that you can repeat the preceding item zero or more times.

`screen font` Represents input examples and screen output examples.

bold font Represents a command name, an option or a complete command line which you can enter.

For example

```
command [option]... filename
```

This line could be written in plain English as: execute the command *command* with the optional options *option* and with the file *filename*.

Illustrations

The following illustrations are used in this manual:

📌 This is a note. It gives you extra information.

⚠ This is a warning. Read the information carefully.

This illustration indicates actions you can perform with the mouse.

This illustration indicates keyboard input.

This illustration can be read as “See also”. It contains a reference to another command, option or section.

Terminology Conventions

”EDE” is used as a shorthand notation for the TASKING Embedded Development Environment, a Windows based integrated development environment for embedded development tools..

CHAPTER

OVERVIEW

1

CHAPTER 1

1.1 EDE PROGRAM DEVELOPMENT

The TASKING Embedded Development Environment EDE can be divided into three main parts:

1. Edit / Project management
2. Build
3. Debug

The figure below shows how these parts interface with each other.

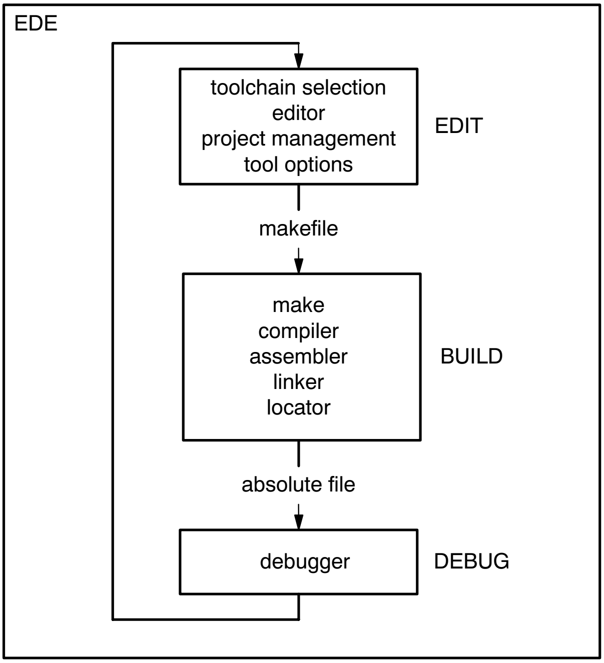


Figure 1–1: EDE development flow

In the **Edit** part you make all your changes. In this part you:

- create and maintain a project
- edit the source files in a project
- set the options for each tool in the toolchain
- select another toolchain if you want to create an application for another target

In the **Build** part a makefile (created by the Edit part) is used to invoke the needed toolchain components, resulting in an absolute object file. In the **Debug** part you can use this absolute file to debug your project.

1.2 WORKING IN AN EMBEDDED ENVIRONMENT

The TASKING EDE differs from a native program development; a native program development is often used to develop applications for systems where the host system and the target are one. Therefore, it is possible to run a compiled application directly from the IDE.

In an embedded environment this is no longer true. Of course you can still compile a module and make it compile error free. However, to run an application, a simulator or target hardware is required. Altium offers a number of simulators and target hardware debuggers. The generic name of the debugger product is CrossView Pro.

CHAPTER

2

GETTING STARTED WITH EDE

TASKING

CHAPTER 2

2.1 STARTING EDE

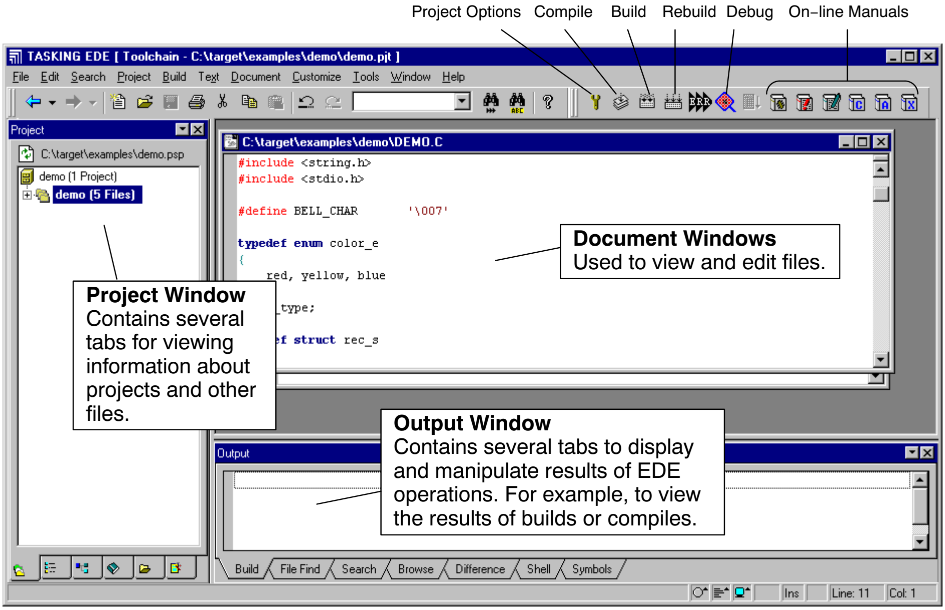
You can launch EDE by double–clicking on the EDE shortcut on your desktop. Or you can launch EDE via the program folder created by the installation program (**Start –> Programs –> TASKING *toolchain* –> EDE**).

2.2 EDE OVERVIEW

EDE is an integrated software development platform that combines a powerful editor, project manager and a make facility. EDE supports all TASKING tools for all targets and is at the same time designed to be open and extensible (i.e. integrate with third party tools). EDE helps you to develop your embedded application by providing the following:

- Full function editor
- Project manager for creating and maintaining projects and project spaces
- Integrated make facility for building your application
- Dialogs to set development tool options for each tool in the toolchain
- Open and extendable environment
- Python–based scripting facility to transfer and control data with third party tools and target–specific code components (for example to integrate Matlab with the CrossView Pro debugger)
- On–line manuals
- Technical support attendant to ease the communication between you and Altium's support engineers.

The EDE screen contains a menu bar, a toolbar with command buttons, one or more windows (default, a window to edit source files, a project window and an output window) and a status bar.



Project Options
Compile
Build
Rebuild
Debug
On–line Manuals
TASKING EDE [Toolchain - C:\target\examples\demo\demo.pjt]
File Edit Search Project Build Text Document Customize Tools Window Help
Project
C:\target\examples\demo.psp
demo (1 Project)
demo (5 Files)
C:\target\examples\demo\DEMO.C
Output
Build
File Find
Search
Browse
Difference
Shell
Symbols
Ins
Line: 11
Col: 1

```
#include <string.h>
#include <stdio.h>

#define BELL_CHAR       '\007'

typedef enum color_e
{
    red, yellow, blue

_type;

ef struct rec_s
```

Document Windows
Used to view and edit files.

Project Window
Contains several tabs for viewing information about projects and other files.

Output Window
Contains several tabs to display and manipulate results of EDE operations. For example, to view the results of builds or compiles.

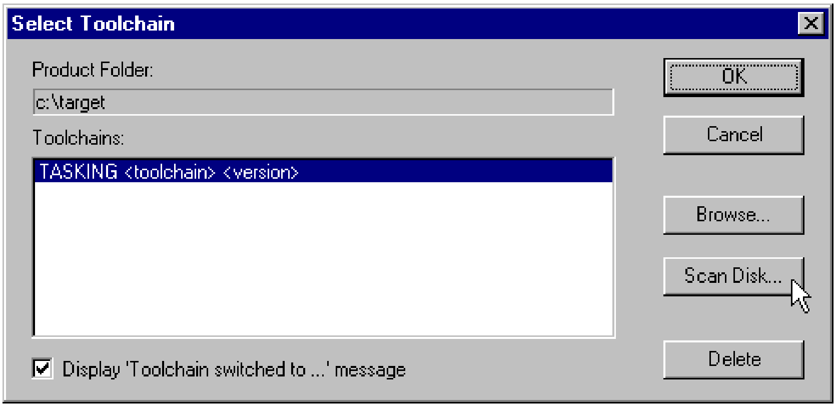
2.3 SELECTING A TOOLCHAIN

EDE supports all the TASKING toolchains. When you first start EDE, the toolchain of the product you purchased is selected and displayed in the title of the EDE desktop window.

If you have more than one TASKING product installed and you want to change toolchains, do the following:

1. From the **Project** menu, select **Select Toolchain...**

   The Select Toolchain dialog appears.



Select Toolchain

Product Folder: c:\target

Toolchains: TASKING <toolchain> <version>

☑ Display 'Toolchain switched to ...' message

2. Select a toolchain in the **Toolchains** list box and click **OK**.

   If no toolchains are present, use the **Browse...** or **Scan Disk...** button to search for a toolchain directory. Use the **Browse...** button if you know the installation directory of another TASKING product. Use the **Scan Disk...** button to search for all TASKING products present on a specific drive.

2.4 PROJECT MANAGEMENT

EDE is a complete project environment in which you can create and maintain project spaces and projects. EDE gives you direct access to the tools and features you need to create an application from your project.

A *project space* holds a set of projects and must always contain at least one project. Before you can create a project you have to setup a project space. All information of a project space is saved in a *project space file* (`.psp`):

- a list of projects in the project space
- history information

Within a project space you can create *projects*. Projects are bound to a target! You can create, add or edit files in the project which together form your application. All information of a project is saved in a *project file* (`.pjt`):

- the target for which the project is created
- a list of the source files in the project
- the options for the compiler, assembler, linker and debugger
- the default directories for the include files, libraries and executables
- the build options
- history information

When you build your project, EDE handles file dependencies and the exact sequence of operations required to build your application. When you push the **Build** button, EDE generates a makefile, including all dependencies, and builds your application.

When EDE generates a makefile, long filenames with spaces are surrounded by double quotes (").

Overview of steps to setup and build an application

1. Create a project space
2. Add one or more projects to the project space
3. Add files to the project
4. Edit the files
5. Set development tool options
6. Build the application

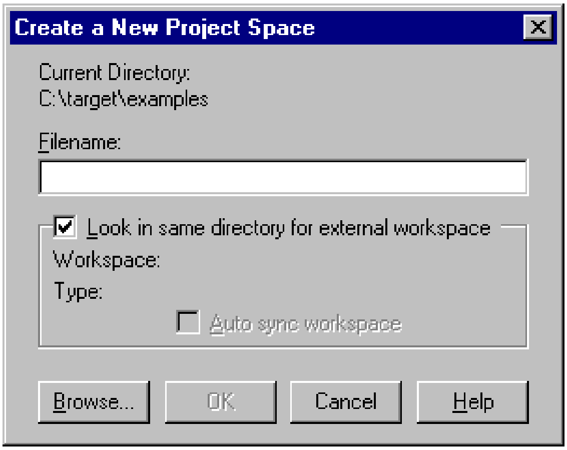
2.4.1 CREATE A NEW PROJECT SPACE WITH A PROJECT

Creating a project space is in fact nothing more than creating a project space file (`.psp`) in an existing or new directory.

Create a new project space

1. From the **File** menu, select **New Project Space...**

   The Create a New Project Space dialog appears.



Create a New Project Space

Current Directory:
C:\target\examples

Filename:

☑ Look in same directory for external workspace

Workspace:

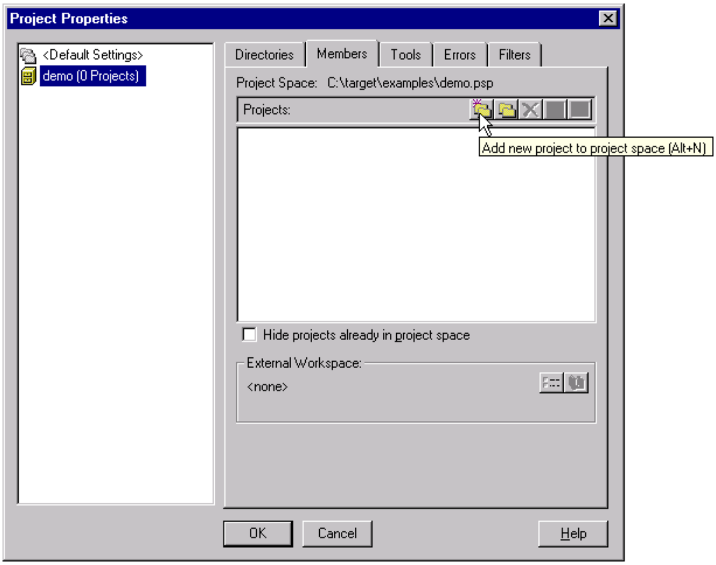
Type:

☐ Auto sync workspace

2. In the **Filename** field, enter a name for your project space (for example `demo`). Click the **Browse** button to select a directory first and enter a filename.

3. Click **OK**.

   A project space information file with the name `demo.psp` *is created and the Project Properties dialog box appears with the project space selected.*



Project Properties

<Default Settings>

demo (0 Projects)

Project Space: C:\target\examples\demo.psp

Projects:

Add new project to project space (Alt+N)

□ Hide projects already in project space

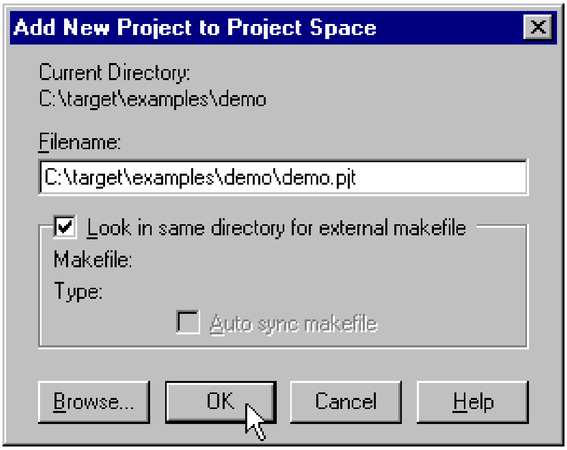
External Workspace:

<none>

Add a new project to the project space

4. In the Project Properties dialog, click on the **Add new project to project space** button.

   The Add New Project to Project Space dialog appears.



Add New Project to Project Space

Current Directory:
C:\target\examples\demo

Filename:
C:\target\examples\demo\demo.pjt

☑ Look in same directory for external makefile

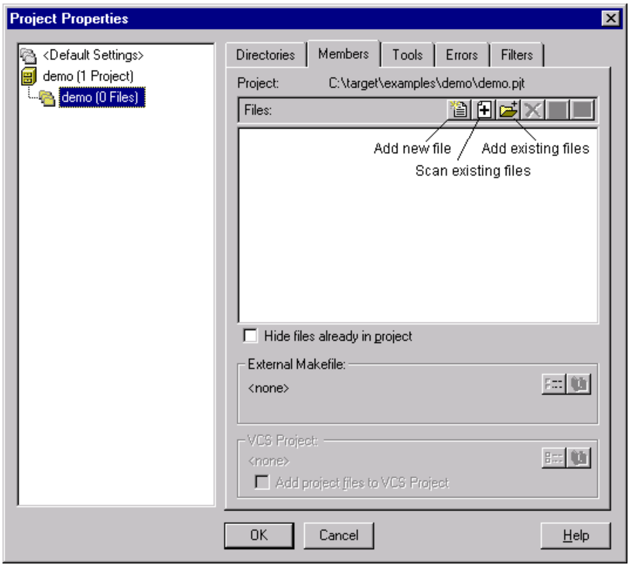
Makefile:

Type:

□ Auto sync makefile

5. Give your project a name, for example `examples\demo\demo.pjt` (a directory name to hold your project files is optional) and click **OK**.

A project file with the name `demo.pjt` *is created and the Project Properties dialog box appears with the project selected.*



Project Properties

<Default Settings>

demo (1 Project)

demo (0 Files)

Directories | Members | Tools | Errors | Filters

Project: C:\target\examples\demo\demo.pjt

Files:

□ Hide files already in project

External Makefile:

<none>

VCS Project:

<none>

□ Add project files to VCS Project

Add files to the project

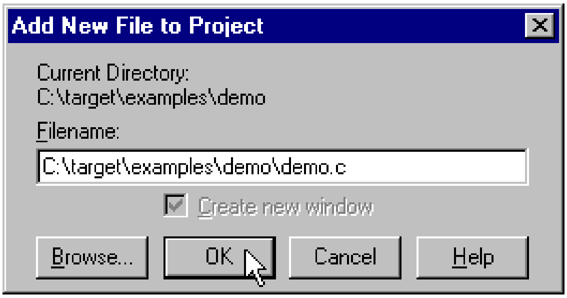
6. Add all the files you want to be part of your project using one of the 3 methods described below.

Method 1: Add new files to a project

Use this method if you want to create a new source file and add it to your project.

a. Click on the **Add new file to project** button.

The Add New File to Project dialog appears.



Add New File to Project

Current Directory:
C:\target\examples\demo

Filename: C:\target\examples\demo\demo.c

☑ Create new window

b. Enter a new filename and click **OK**.

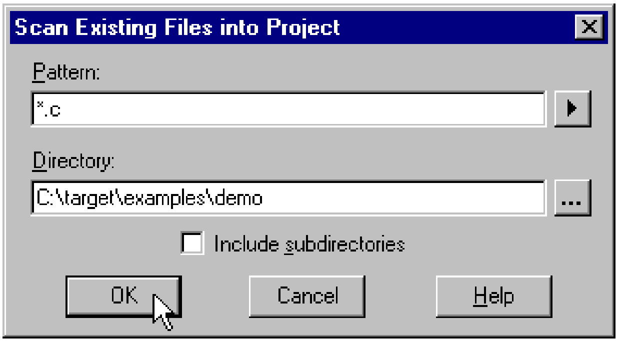
A new empty file is created and added to the project.

Method 2: Scan existing files into a project

Use this method to add existing files to a project by specifying a file pattern. For example, to add all C files in a specific directory to your project.

a. Click on the **Scan existing files into project** button.

The Scan Existing Files into Project dialog appears.



Scan Existing Files into Project

Pattern: *.c

Directory: C:\target\examples\demo

□ Include subdirectories

b. In the **Directory** field, type or select the directory that contains the files you want to add to your project.

c. In the **Pattern** field, enter one or more file patterns separated by semicolons. Click the button next to the **Pattern** field to select a predefined pattern.

d. Click **OK**.

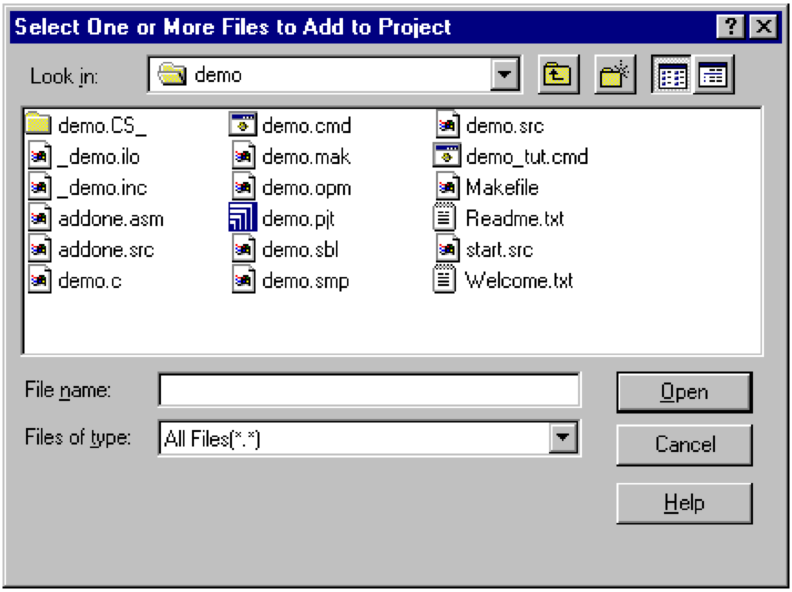
The files that match the specified pattern(s) are added to your project.

Method 3: Add existing files to a project

Use this method to add existing files to a project by selecting individual files.

a. Click on the **Add existing files to project** button.

The Select One or More Files to Add to Project dialog appears.



Select One or More Files to Add to Project

Look in: demo

demo.CS_
_demo.ilo
_demo.inc
addone.asm
addone.src
demo.c
demo.cmd
demo.mak
demo.opm
demo.pjt
demo.sbl
demo.smp
demo.src
demo_tut.cmd
Makefile
Readme.txt
start.src
Welcome.txt

File name:

Files of type: All Files(*.*)

b. In the **Look in** box, select the directory that contains the files you want to add to your project.

c. Select the files you want to add (hold down the **Ctrl**–key to select more than one file) and click **Open**.

All the selected files will be added to your project.

7. Click **OK**.

The new project is now open.

EDE automatically creates a *makefile* for the project. This file contains the rules to build your application. EDE updates the makefile every time you modify your project.

2.4.2 ADDING PROJECTS TO AN EXISTING PROJECT SPACE

When you already have a project space (your own project space or one of the delivered examples), you can either add a new project or add an existing project to the open project space.

Open an existing project space

1. From the **File** menu, select **Open Project Space..**

   The Select Project Space to Open dialog appears.

2. Select a project space and click **Open**.

   The project space (and its projects) appears in the Project Window.

Add a new project to a project space

1. From the **File** menu, select **Configure Project Space –> Add New Project...**

   The Add New Project to Project Space dialog appears.

2. Follow steps 5 to 7 described in the previous section (2.4.1).

Add existing projects to a project space

1. From the **File** menu, select **Configure Project Space –> Add Existing Projects...**

   The Select One or More Projects to Add to Project Space dialog appears.

2. Select the project files you want to add (hold down the **Ctrl**–key to select more than one project) and click **Open**.

   The Project Properties dialog appears.

3. Click **OK**.

Include a sub–project

Projects not only contain files, but can also contain *sub–projects*. This can be useful for including libraries in a project: you can include the project that builds the library as a sub–project in the project that uses the library. Whenever the main project is built, it will first check if the library (the sub–project) is up–to–date. If not, the library is rebuilt before the main project is built.

To include a sub–project:

1. Right–click on the active project in the **Project Window** and select **Include Sub–Project...**

   – or –

   Use *Method 3: Add existing files to a project*, as described in the previous section, and select **TASKING Project Files** as a file type.

2. Select a project file (`.pjt`) and click **OK**.

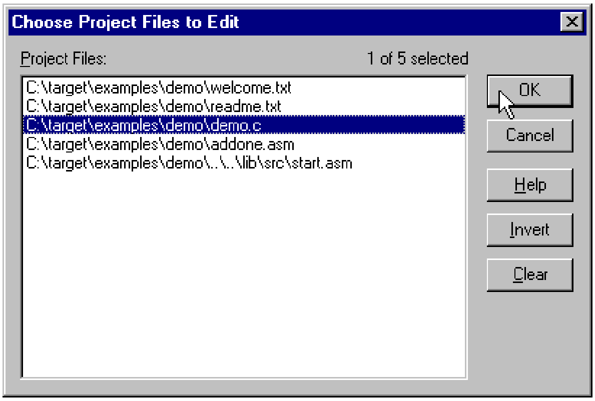
   The selected project is added as a sub–project to your main project.

2.5 HOW TO LOAD/OPEN FILES

The next step in the process of building your application is to open the files you want to edit or look at.

1. From the **Project** menu, select **Load Files...**

   The Choose Project Files to Edit dialog appears.



Choose Project Files to Edit

Project Files: 1 of 5 selected

C:\target\examples\demo\welcome.txt
C:\target\examples\demo\readme.txt
C:\target\examples\demo\demo.c
C:\target\examples\demo\addone.asm
C:\target\examples\demo\..\..\lib\src\start.asm

2. Select the file(s) you want to open (hold down the **Ctrl** key to select more than one file) and click **OK**.

   EDE loads the selected files in the editor in separate document windows.

3. Optionally, edit the file(s).

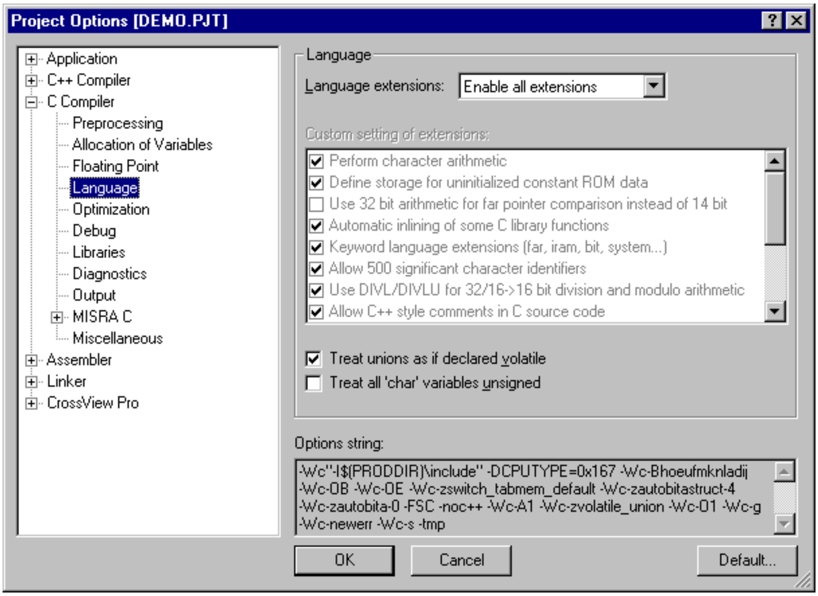
2.6 SETTING DEVELOPMENT TOOL OPTIONS

The next step in the process of building your application is to specify the options for the different parts of the toolchain, such as the C and/or C++ compiler, assembler, linker and debugger.

The screen captures in this section serve as examples, they may look slightly different for your toolchain.

1. From the **Project** menu, select **Project Options...**

   The Project Options dialog appears. This dialog contains several entries where you can specify processor and development tool options.



Language

Language extensions: Enable all extensions

Custom setting of extensions:

☑ Perform character arithmetic
☑ Define storage for uninitialized constant ROM data
☐ Use 32 bit arithmetic for far pointer comparison instead of 14 bit
☑ Automatic inlining of some C library functions
☑ Keyword language extensions (far, iram, bit, system...)
☑ Allow 500 significant character identifiers
☑ Use DIVL/DIVLU for 32/16->16 bit division and modulo arithmetic
☑ Allow C++ style comments in C source code

☑ Treat unions as if declared volatile
☐ Treat all 'char' variables unsigned

Options string:

```
-Wc"-I$(PRODDIR)\include" -DCPUTYPE=0x167 -Wc-Bhoeufmknladij
-Wc-OB -Wc-OE -Wc-zswitch_tabmem_default -Wc-zautobitastruct-4
-Wc-zautobita-0 -FSC -noc++ -Wc-A1 -Wc-zvolatile_union -Wc-O1 -Wc-g
-Wc-newerr -Wc-s -tmp
```

2. For each entry make your changes. If you have made all changes click **OK**.

The **Cancel** button closes the dialog without saving your changes. With the **Defaults** button you can restore the default project options (for the current page, or all pages in the dialog).

If available, the **Options string** field shows the command line options that correspond to your graphical selections.

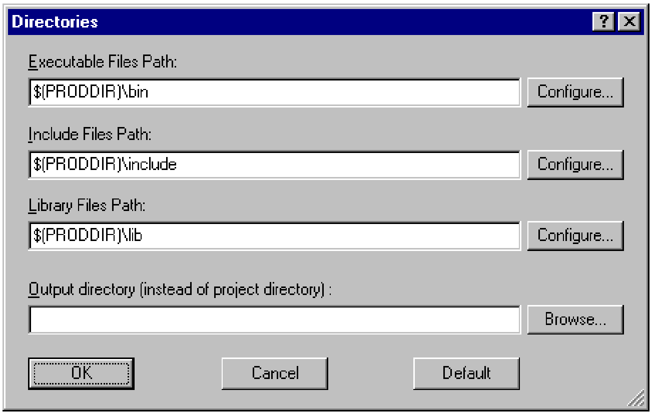
2.7 CHECK THE DIRECTORY PATHS

EDE and the development tools use several directories to find executable programs, include files and libraries. EDE uses default settings for your toolchain. You may want to check or change these defaults.

Check the directory paths

1. From the **Project** menu, select **Directories...**

   The Directories dialog appears.



Directories

Executable Files Path: $(PRODDIR)\bin

Include Files Path: $(PRODDIR)\include

Library Files Path: $(PRODDIR)\lib

Output directory (instead of project directory) :

2. Check the directory paths for programs, include files and libraries. You can add your own directories here, separated by semicolons.

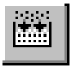
3. Click **OK**.

2.8 HOW TO BUILD YOUR APPLICATION

The next step is to compile the file(s) together with its dependent files so you can debug the application.

Build your Application

- Click on the **Execute 'Make' command** button. The following button is the execute Make button which is located in the toolbar.



If there are any unsaved files, EDE will ask you in a separate dialog if you want to save them before starting the build.

Viewing the Results of a Build

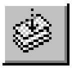
Once the files have been processed, you can see which commands have been executed (and inspect generated messages) by the build process in the **Build** tab of the **Output** window.

This window is normally open, but if it is closed you can open it by selecting the **Output** menu item in the **Window** menu.

Compiling a Single File

1. Select the window (document) containing the file you want to compile or assemble.

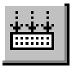
2. Click on the **Execute 'Compile' command** button. The following button is the execute Compile button which is located in the toolbar.



How to Rebuild your Entire Application

If you want to compile, assemble and link/locate all files (regardless of their date/time stamp) of your project from scratch, you can perform a rebuild. This may be needed after dependencies have been changed.

- Click on the **Execute 'Rebuild' command** button. The following button is the execute Rebuild button which is located in the toolbar.

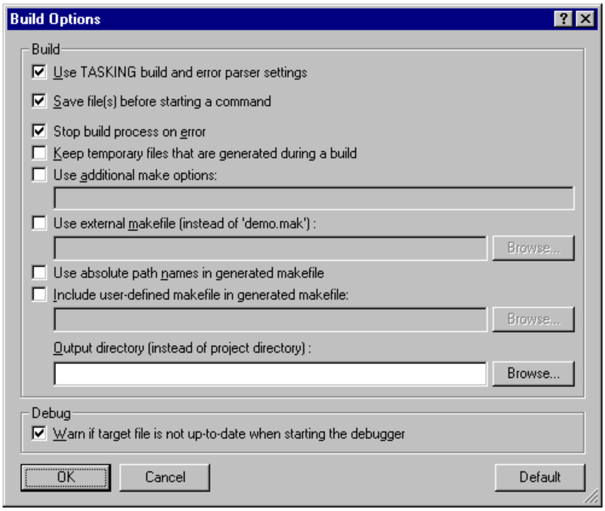


2.8.1 ADVANCED BUILD OPTIONS

These steps are optional. Follow these steps if you want to specify additional build options such as to stop the build process on errors and to keep temporary files that are generated during a build.

1. From the **Build** menu, select **Options...**

   The Build Options dialog appears.



Build Options

Build

☑ Use TASKING build and error parser settings

☑ Save file(s) before starting a command

☑ Stop build process on error

☐ Keep temporary files that are generated during a build

☐ Use additional make options:

☐ Use external makefile (instead of 'demo.mak') :

☐ Use absolute path names in generated makefile

☐ Include user-defined makefile in generated makefile:

Output directory (instead of project directory) :

Debug

☑ Warn if target file is not up-to-date when starting the debugger

All EDE command settings are handled in this dialog, unless you disable the **Use TASKING build and error parser settings** check box. In that case, you should change the Compile, Build, Rebuild and Debug settings in the **Tools** tab of the **Project | Properties...** dialog box.

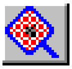
2. Make your changes and click **OK**.

2.9 STARTING THE CROSSVIEW PRO DEBUGGER

Once the files have been compiled, assembled, linked, located and formatted they can be executed by CrossView Pro.

To execute CrossView Pro:

- Click on the **Debug application** button. The following button is the Debug application button which is located in the toolbar.



CrossView Pro is launched. CrossView Pro will automatically download the compiled file for debugging.

See the *CrossView Pro Debugger User's Manual* for details about the debugger.

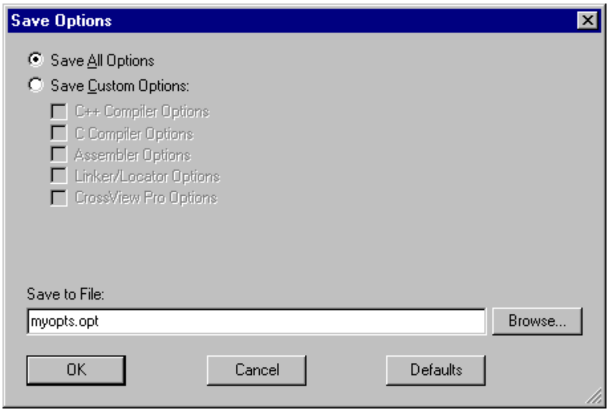
2.10 HOW TO LOAD/SAVE OPTIONS

You can save the settings of your toolchain options into an option file (`.opt`) so that you can restore a specific configuration whenever you want. You can also specify to automatically load an option file when a project is opened.

To save options to a file

1. From the **Project** menu, select **Save Options...**

   The Save Options dialog appears.



● Save All Options
○ Save Custom Options:
□ C++ Compiler Options
□ C Compiler Options
□ Assembler Options
□ Linker/Locator Options
□ CrossView Pro Options

Save to File: myopts.opt

2. Optionally, select **Save Custom Options** and select the tool options you want to save. The default is **Save All Options**.

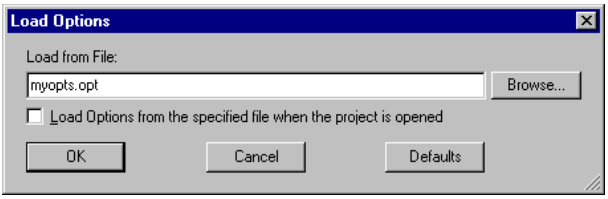
3. Enter a filename and click **OK**.

   All toolchain option settings for the current project are saved in the specified option file.

To load options from a file

1. From the **Project** menu, select **Load Options...**

   The Load Options dialog appears.



myopts.opt

□ Load Options from the specified file when the project is opened

2. Enter the filename of a previously saved option file.

3. Optionally, enable the **Load Options from the specified file when the project is opened** check box.

4. Click **OK**.

2.11 TECHNICAL SUPPORT

EDE has an automated technical support attendant to ease the communication between you and Altium’s support engineers.

The attendant prepares an Email which will be exported to your Email software (you can inspect the mail and all attached files before you transmit them via your Email software). EDE helps you to provide the support engineer with all the information he needs, such as product version and serial numbers, FLEXlm troubleshooting information, and assists you in attaching source files, linker control files and other projects files to your Email.

To prepare an Email for technical support

1. From the **Help** menu, select **Technical Support –> Prepare Email...**

   The Prepare Email dialog appears.

2. Fill in the necessary information, such as serial number, customer information, Email subject, problem description and the Email address of the support desk.

3. Click **Copy to Email client**.

4. Inspect the mail in your Email software and send it.

CHAPTER 3

USE EDE WITH CODEWRIGHT

TASKING

CHAPTER 3

3.1 INTRODUCTION

Instead of using the editor delivered with EDE (which is also based on CodeWright technology) you can use EDE with the popular CodeWright editor. Therefore, we deliver EDE as a dynamic link library (`EDE32.DLL` for Windows 95/98/XP/NT/2000) which can be loaded by your existing CodeWright configuration at startup. All you have to do is:

1. Add two lines to the `LibPreload` section of the configuration file `CWRIGHT.INI`, which is part of your CodeWright installation:

   ```
   [LibPreload]
   LibPreload=ede32.dll
   LibPreload=cwdde.dll
   ```

   In your `CWRIGHT.INI` file the line

   ```
   ;LibPreload=cwdde.dll
   ```

   may already be present, but as comments. In this case you only have to remove the semicolon at the start of the line.

2. Either copy `EDE32.DLL` to the CodeWright directory or specify the full path (replace *product* with the name of your toolchain directory):

   ```
   LibPreload=C:\product\bin\ede32.dll
   ```

   Use the `CWDDE.DLL` which is part of your CodeWright product to prevent possible incompatibilities.

3. Optionally, if the file `EDESRC.DLL` exists in your toolchain `bin` directory, you have to add one extra line to the `LibPreload` section of the configuration file `CWRIGHT.INI`:

   ```
   LibPreload=C:\product\bin\edesrc.dll
   ```

We recommend NOT to use the environment variables CWINI and CWLIB. However, in case of a network installation of a TASKING product or CodeWright, you have to set these environment variables. Please check that CWINI points to the correct `CWRIGHT.INI` file and that CWLIB contains the directory where the DLLs are.

3.2 FILES USED BY EDE

The following files are used by EDE:

`\`*product*`\bin\ede32.dll`	EDE program as a 32–bits DLL
`\`*product*`\bin\edesrc.dll`	EDE DLL delivered with some toolchains to provide toolchain specific ChromaCoding
`\`*product*`\bin\wmk.exe`	TASKING Make utility (32–bits)
`\`*product*`\bin\cwright.ini`	Configuration file

CHAPTER

EDITOR

4

TASKING

CHAPTER 4

4.1 INTRODUCTION

EDE is based on CodeWright technology. For a detailed description of the editor refer to the on–line manual of the editor: *CodeWright User's Manual.*

4.2 HELP

EDE offers several types of Help services:

Context–sensitive Help
Provides you with information about dialog boxes and other topics as you are using EDE. You access this feature by pressing `F1`.

EDE API Help
Provides information about the functions in the EDE API. Place the cursor in the word or function name for which you want help, and press `CTRL F1`.

Windows API Help
Accesses the SDK or other help files to provide information about the Windows API. As above, place the cursor in the word or function name for which you want help, and press `CTRL F1`.

Scanned Topics Index
Allows you to scan help files, adding the topics they contain to an index of known help topics. When you press `CTRL F1` CodeWright will scan the index for the word at the cursor. See the `Configure Index File` on the `Help` menu.

EDE help
This manual.

Toolchain help
You can access all TASKING manuals also via icons on the toolbar or via the `Help` menu.

4.3 PRACTICAL TIPS

Keystroke Macros Dialog

You can record keystrokes and play them back by pressing the `F7` and `F8` keys respectively. On the `Edit` menu, you will find the `Keystroke Macros` dialog and the `Record` and `Playback` items. The `Keystroke Macros` dialog provides a powerful extension of this capability. You copy keystroke recordings to and from the numbered storage areas listed in this dialog. These recording are saved and restored between sessions, and you can also give each recording a description so you will know what it does later.

Document/Window Manager Dialog

You can access the Document/Window Manager dialog by selecting `Manager...` from the `Document` menu or from the `Window` menu. Several tabs are available.

Below is a list of questions to ask yourself about the settings you will find in this dialog. When it comes to act on your answers, press `F1` if you need further description of the options available:

- Do I want to use tab characters in my buffers or do I want to use the equivalent spaces? Spaces are not open to interpretation. Tabs are more flexible.
- Do I want to make backup files? (If so, the best place to set the name of the backup file in the system Options dialog. The default is `.BAK`)
- Do I want Auto–indent? Which fill mode? (Consider using Seek Indentation.)
- Do I want the cursor to able to travel past the end of the buffer?
- Do I want scroll bars on my document windows?
- Do I want line numbers displayed at the left edge of the window?
- How much space do I want between the left window border and the beginning of text? (More space makes line operations with the mouse easier and gives more room for Changed–line Marking)
- Do I want any visible representation for invisible aspects of the displayed buffer? Tabs? End of line? End of file? (It is easy to over do this stuff and get an ugly display. See the Visible function for occasional use.)

Language Dialog

This is a place where a lot of the ”good stuff” is hiding. You turn on ChromaCoding from here, Language Templates and other bells and whistles.

Tab Setting and Examples

This is one of several ways to set up tabs for buffers. This is used for all new buffers that bear the extension in the File Type edit box. The method of specifying tabs is the same throughout CodeWright. You specify the interval between tab stops by specifying one or more columns at which there are tab stops.

For example, the string ”5 9” has an interval of 4 between the two tab stops specified. CodeWright therefore not only puts stops at columns 5 and 9, it repeats the tab stops at four column intervals all the way out to column 200. Actually, a tab setting of ”5” will do the same thing. In this case, CodeWright looks at the interval between the one tab stop specified and column 1. One last example ”8 12” skips over columns 2 through 7 and thereafter repeats at an interval of 4.

ChromaCoding

This is where you go to turn on that ChromaCoding feature syntax Highlighting. You need to turn it on for each file type you will be using. You set the actual colors used in another place –– Colors on the Window menu. Of course, Language Dependent ChromaCoding only works for languages for which there is built–in support.

Smart Indenting

Smart Indenting simply means that if you type certain words and then press enter, CodeWright will increase the indentation on the newly created line. Many users find that the indentation built into templates combined with language indentation is just too darned much indentation. For this reason you may select any combination of templates and language indentation.

Environment Dialog

Let us consider the settings available under the Environment dialog. Select `Environment` from the `Customize` menu. There are a number of things here you want to overlook. Remember, for more details about these options or other Environment options just press `F1`. Here are some questions worth asking:

- Do you want system prompts to pop–up in a dialog, or to be displayed on the status line? For most people, the status line has some advantages.
- Do you want CodeWright to remember and load just the last file you were working on or all files? Maybe you do not want it to remember anything between sessions.
- Do you want windows and buffers to operate as a single unit, like they do in most Windows applications, or independently?
- Do you want a list of recently loaded files to appear at the File menu? This can be quite handy for reloading files.
- Do you want to use the Tool Ribbon, the SideBar or Both? Note that when you pass the mouse over the buttons a help message appears describing the button. At some point this may become distracting. If so, this is the place to turn off those messages.

Auto–save

You will find Auto–save on the `Backup` tab of the `Environment` dialog. If you will be building and running Windows applications from within CodeWright, auto–save can be an especially important feature. Not that a program of yours would ever bring down Windows, but strange things can happen when you are programming. It is best not to have too much unsaved data in your editor at times like these.

APPENDIX A

PYTHON

APPENDIX

A

1 INTRODUCTION

You can configure EDE to use Python scripts. The purpose of integrating the Python interpreter with EDE is to control EDE or external applications by using Python scripts. This is useful to connect multiple CrossView Pro debuggers with arbitrary third–party tools, like Matlab, RiMC or Consystant.

2 INSTALLATION AND CONFIGURATION

In order to use Python with EDE you must have the Python 2.0 interpreter installed. If you want to use COM, for example to control Matlab, you must also install the Win32 extensions.

Follow these steps to install and add Python to EDE:

1. Install the Python 2.0 interpreter. You can use the Python 2.0 installation program `beopen–python–2.exe` in the `bin\python` directory of your product.

   If you want to use COM, also install the Win32 extensions. You can use the installation program `win32all–135–py20–compatible.exe` in the `bin\python` directory.

2. One loadable DLL (`cwpythoni.dll`) is necessary to use Python with EDE. To enable this DLL, select `Customize | Libraries` and select `Python Extension Language`. If the Python interpreter is not present in the predefined libraries list, Browse for a User Defined Library named ”`cwpythoni.dll`”.

   If the Output Window is not already showing, select it from the Window menu. You should then see a `Python` tab on the Output Window. This window acts as a virtual console for the Python interpreter; that is, it replaces stdin, stdout and stderr in UNIX parlance.

2.1 POPUP MENU

If you right–click on the `Python` tab a Python popup menu appears that offers various options for the Python language.

Clear Window

Erases the contents of the Python Output Window.

Horiz. Scrollbar

Toggles the presence of a horizontal scrollbar the Python Output Window.

Run...

Executes a Python script, selected using the `Select Python Script to Load` dialog which appears. The directory where this dialog starts is always the current project's directory.

Hint: By removing the filename of the path, and typing Enter, the directory is changed to the last script's directory.

Control–break

Stops the running Python script. The interpreter is limited to detecting the halt request when doing I/O, that is reading from or printing to the Python Output Window.

Properties...

Opens the Python interpreter EDE Python Options dialog.

2.2 EDE PYTHON OPTIONS

The EDE Python Options dialog allows you to configure the Python interpreter's command line options (`Properties` and `Other Interpreter Invocation Options`), and you can specify how standard I/O is connect to EDE (`Input Source` and `Output Destination`).

You can extend the module search path in the `Additional Python Library Directories` field. Add the directories you want to be searched for Python scripts and modules. Since the Python interpreter loads the library once at startup, you must restart EDE to effect the changes to this entry.

There is a limit on how many lines of output will be retained in the Python Output Window. You can change this limit in the `Python Window Line Limit` field.

3 RUNNING PYTHON SCRIPTS

The Python extension adds API calls to EDE which you can issue as regular API function calls. To give an API command, enter the command via the `Tools | API Command` dialog.

To execute a Python script, use the `PythonExec` API call or right–click on the `Python` Output Window, select `Run...` and specify a Python script.

To execute a single Python statement, use the `PythonExecStr` API call.

To stop a running Python script, right–click on the `Python` Output Window and select `Control–break`.

The following API calls are present:

PythonExec *script_fle any_parameters*

Executes a Python script. *script_file* is the name of the Python script (usually with the `.py` extension), *any_parameters* consists of any parameters required by the Python script.

Example:

```
PythonExec C:\test.py
```

PythonExecStr *statement*

Executes a Python statement. *statement* is any Python statement.

Example:

```
PythonExecStr print 3+4
```

PythonCtrlBreak

Stops the running Python script. The interpreter is limited to detecting the halt request when doing I/O, that is reading from or printing to the EDE Python window. This API call intended for use under a toolbar button.

4 FILES USED BY PYTHON FOR EDE

The following files are present in the `bin` directory:

cwpythoni.dll

Provides language support for Python scripts and adds the `Python` tab to the Output Window.

cwpythoni.mnu

Menu file that holds the menus for the right mouse click in the `Python` tab of the Output Window.

_ddeclt.py

A strings based DDE client module, including Advise hot–link support. The module does not rely on MFC.

ddeclt.pyd

An OO wrapper around _ddeclt, to ensure `Uninitialize()` invocation.

py2cwapi.pyd

Allows calling many of the EDE API functions.

redir2cw.py

Binds Python STDIO to EDE and provides control and API calling.

matlab.py

Python object wrapper around the ActiveX interface of Matlab.

5 DDECLT PYTHON EXTENSION MODULE

ddeclt is a strings data based DDE client. A module which does not rely on MFC. It provides a DDE service object class DDEclient, which must be instantiated to enable using the DDE API.

The DDEclient object is a Python wrapper around a C module, `_ddeclt`, which provides the C API binding into Python.

Example:

```
import ddeclt

dde = DDEclient()
dde.Connect(’application’, ’topic’)
dde.Execute(’command’)
del dde
```

Known problems

1. Request fails while an Advise link is present.

   When an item is attached to with a hot link, see `Advise()`, a `Request()` may fail when the server frequently updates the hot linked item. Somehow the `Request`ed data gets invalidated by the recursive servicing of the Advise link. Apparently the cause is the XTYP_ACKREQ flag to the `Advise()` call. For Windows NT it is not necessary, hence the default is off.

2. The receiver does not get the messages sent to it.

   This happens if the message queue fills up under Windows 3.1 and 9x. If the client does not keep up with the server, the message queue will eventually fill up. Then messages are lost because `PostMessage` fails. The only answer is to stop the server manually when (and if) prompted by Windows. The user interfaces will probably not respond. If the server continues attempting to post messages the system is likely to crash. The `fAckReq` flag was introduced to address this problem.

   Under Windows NT this does not happen, the queue continues to grow. If the client catches up, the queue starts reducing again. If not, the task scheduling is changed to reduce it. The user interfaces will probably continue to respond, although the Task Manager may report that the application is not responding. Windows NT appears to be bullet–proof, you can safely choose not to use the `fAckReq` flag.

3. Failure to handle errors when using synchronous transactions.

   DDE is an asynchronous process, but is made to appear synchronous when the message sender chooses to suspend execution while waiting for a reply. If the timeout period specified is not long enough, the transaction will appear to have failed. Further problems may arise when the message from the server finally arrives. A much better way is to work asynchronously, a callback routine is entered when the partner application replies.

   So far, the **ddeclt** module does only provide synchronous operations. The Python interpreter's callback mechanism was not suitable yet, limiting callback requests to a fixed 32 (see `Py_AddPendingCall()`). You can, however, use an Advise link to receive asynchronously and use the CrossView Pro command **execnowait**: to emulate asynchronous execution.

5.1 DDE CLIENT METHODS

ProcessAllPendingMessages(first_msg = 0, last_msg = 0)

Process (pump) all waiting messages for the current thread.

Returns 1 if a WM_QUIT event was received, 0 otherwise.

WaitForAndProcessOneMessage()

Process (pump) one waiting message for the current thread. It is necessary for a DDE client to process events, to service the Advise link. This function is useful when using a custom DDE callback function in Python. See `Initialize()`.

Returns 1 if a WM_QUIT event was received, –1 in case of error, 0 otherwise.

GetAdvisedItemNoWait(*connection, itemname****)***

Returns a tuple (*value*, *data–event*) where *value* is the data received accompanying a data changed Advise event, and d*ata–event* is a boolean, which is true when a data changed event has been received. This event is to be used in case `use_NODATA` has been specified to `Advise()`, signalling data can be requested from the server. If no data change event has been received `None` is returned. Resets the internal change flag used by `WaitForAnyAdvisedItem()`, so check for all necessary items. This function does not wait for a data change Advise event. If a callback has been specified to `Initialize()`, this function is not operational.

If Advise has been invoked with `use_NODATA=1`, a tuple with `None` as first member value, instead of a string, is returned if the server sent a notification. The data is to be retrieved using `Request()`.

Request(*connection, item[, timeout]****)***

Request data from a server. The timeout is default set to 25 days, and has to be specified in milliseconds. The timeout cannot be disabled.

Returns the data received.

ConversationHandle(*connection****)***

Returns the DDE conversation handle of the specified connection tuple.

Unadvise(*connection, item[, timeout]****)***

Remove a data change Advise loop of the specified item.
The timeout is default set to 25 days, and has to be specified in milliseconds.
The timeout cannot be disabled.

Returns nothing.

GetServiceTopicsList(*service_name****)***

Returns a list of topics supported by the specified service (aka server application).

Sets up and terminates its own conversation for interrogation purposes. The timeout is fixed at 5 seconds.

Uninitialize()

Frees all Dynamic Data Exchange Management Library (DDEML) resources associated with the specified application.

Returns nothing.

Also implicitly done when deleting the DDEclient object.

Disconnect(*connection****)***

Terminates a conversation started by either the `DdeConnect` or `DdeConnectList` function and invalidates the specified conversation handle. Any incomplete transactions started before calling `DdeDisconnect` are immediately abandoned.

Returns nothing.

Connect(*service_name, topic_name****)***

Establishes a conversation with a server application that supports the specified service name and topic name pair. If more than one such server exists, the system selects only one.

Returns a connection tuple to be used as argument to most other DDECLT function calls.

WaitForAnyAdvisedItem()

Wait forever for the next data changed Advise event for any item. However, if any data is still pending this function immediately returns. If not, only terminates prematurely when a WM_QUIT event is received.

Returns 1 if a WM_QUIT event was received, else 0.

If a callback has been specified to `Initialize()`, this function is not operational.

GetItemsList(*service_name, topic_name****)***

Returns a list of items supported by the specified service and topic pair.

Sets up and terminates its own conversation for interrogation purposes. The timeout is fixed at 5 seconds.

Execute(*connection, command_string[, timeout]****)***

Send a command to the connected DDE peer, usually a server. The timeout is default set to 25 days, and has to be specified in milliseconds. The timeout cannot be disabled.

Returns nothing. If the DDE exchange fails, e.g. if the server is busy, an exception occurs.

Initialize( )

Registers an application with the Dynamic Data Exchange Management Library (DDEML). An application must call this function before calling any other DDEML function.

See `WaitForAndProcessOneMessage()` for processing DDE events when waiting inside Python.

Returns the application instance identifier handle, abbreviated here to DDEML lib handle.

Also implicitly done when create–ting the DDEclient object.

GetAdvisedItem(*connection, itemname****)***

Wait forever for the next data changed Advise event of the specified item. Only terminates prematurely when a WM_QUIT event is received. Note that other items may also be received while waiting.

Returns a tuple (*value*, *data–event*) where *value* is the data received accompanying a data changed Advise event, *data–event* is a boolean, which is true when a data changed event has been received. This event is to be used in case `use_NODATA` has been specified to `Advise()`, signalling data can be requested from the server. If no data change event has been received `None` is returned. If a callback has been specified to `Initialize()`, this function is not operational.

ConnectList(*service_name, topic_name****)***

Establishes a conversation with all server applications that support the specified service name and topic name pair. By specifying empty strings for either service_name or topic_name, all possible combinations are connected to.

Returns a list of connection tuples, to be used with for example `Request` or `Disconnect`.

Advise(connection, item[, timeout]][, use_ACKREQ=0][, use_NODATA=0])

Establish an advise loop with a server.

The server will send data changed Advise events when new data is present. If an advise loop already is present for the specified *item*, the existing advise loop will be replaced.

The *timeout* is default set to 25 days, and has to be specified in milliseconds. The timeout cannot be disabled.

`use_ACKREQ` has the server wait for us to tell we are ready for the next data. Apparently required for older Windows versions, not for NT. See below. Using this option may cause Advise events to get lost.

`use_NODATA` tells the server to hold back the data, just send the data changed notification event. Obtaining the data with a request from inside the callback (see `Initialize`), causes reentrancy problems.

Returns nothing.

Since Python does not support Windows–style asynchronous callbacks, neither via `Py_AddPendingCall()` because of its fixed size buffer, Advise data is stored in a linked list, which is emptied via one of `GetAdvisedItem()`, `GetAdvisedItemNoWait()`. Hence, regularly call one of these, to flush the underwater buffer.

WaitForAndProcessAllMessages()

Process forever (pump) all messages for the current thread. Only returns on WM_QUIT or error. It is necessary for a DDE client to process events, to service the Advise link. DDE Advise must be serviced, otherwise it blocks the process's execution. Apparently the DDE callback simply runs inside the process, on top of its stack.

Returns 1 if a WM_QUIT message was received, else 0 signalling an error occurred in `GetMessage()`.

Poke(connection, item, value[, timeout])

Send unsolicited data to the server The timeout is default set to 25 days, and has to be specified in milliseconds. The timeout cannot be disabled.

Returns nothing.

6 PY2CWAPI

This module wraps most of the EDE API functions, for direct usage from Python.

The number of API functions is far too big to be included here though. Check the directory, using the Python built–in function `dir(py2cwapi)`, for all functions provided by this module. Start Python in the directory `bin` directory of toolchain (the directory where `py2cwapi.pyd` is located). A prompt appears. Type `import py2cwapi` and then `dir(py2cwapi)`.

```
>>> import py2cwapi
>>> dir(py2cwapi)
>>> a list of functions appears
```

See the API library description in EDE for each function's features.

Example:

```
from py2cwapi import *

ZoomWindow()
```

7 REDIR2CW

ConsoleHorScrollBar(request)

Determine the new state of the scrollbar, or query its current state.

Value	Function
>0	Turn on the scrollbar.
0	Turn off the scrollbar.
–1	Return a value >0 indicating if the scroll bar is on, else 0.
–2	Toggle the scrollbar state.
<–2	No op.

In all cases except –1, the return value indicates whether or not the scrollbar was on prior to the call.

SelectIODevice(input_channel_nr, output_channel_nr)

Select where STDIO is connected to in EDE.

Channel numbers:

cw_console
cw_current_document_buffer
cw_selected_text
cw_clipboard
cw_current_scrap_buffer

Default STDIO is connected to cw_console.

Returns the previously selected I/O device numbers in a tuple (*input*, *output*).

ClearConsole( )

This function erases the content of the Python console window.

PollControlBreak( )

Polls the EDE GUI, without any time interval. If control break is hit, an `KeyboardInterrupt` exception is raised. This function is necessary because the Python interpreter does not provide a hook to do this internally.

CallCwFunction(*function_name [,arg...]* ***)***

Invoke an EDE API function. Conversions are done under water. The function name and arguments are expected in a tuple.

ATTENTION: Python cannot specify C type characters, so specify an integer value instead.

Returns the value returned by EDE.

See also `LibFunctionExec` and `LibExport` in the EDE API manual.

Example:

```
from redir2cw import *

CallCwFunction(”ZoomWindow”)
```

CwConstantValue(*constantname****)***

Returns the value of a EDE constant

Example:

```
from redir2cw import *

print CwConstantValue(”EXPR_FCT_ARGS”)
```

8 MATLAB VIA ACTIVEX WRAPPER CLASS

A Python module has been included which declares the class `Matlab`, wrapping Matlab instantiation and execution.

You can test the Matlab connection by running the module as a Python script.

8.1 CLASS MATLAB METHODS

GetSingleton(*variablename*, ***workspace*** = *'base'****)***

Read a single value from a Matlab variable.

PutSingleton(*variablename*, *value*, ***workspace*** = *'base'****)***

Write a single value to a Matlab variable.

GetMatrix(*matrixname*, ***columns*** **= *1*, *rows*** **= *1*, *workspace*** = *'base'****)***

Get a real matrix.

The number of columns and rows of the matrix must be specified. Always returns a matrix, even if it has one row. The value is a list of lists, with each sub–list being one row.

GetMatrixOfSameSize(*matrixname*, *dummy_matrix*, ***workspace*** = *'base'****)***

Returns a real matrix, as `GetMatrix`, but instead of specifying the dimensions, pass a matrix with the expected dimensions. Its value will not be changed.

Always returns a matrix, even if it has one row.

PutMatrix(*matrixname*, *real*, ***workspace*** = *'base'****)***

Put a real matrix. The value is a list of lists, with each sub–list being one row.

GetComplexMatrix(*matrixname*, ***columns*** **= *1*, *rows*** **= *1*, *workspace*** = *'base'****)***

Get both the real as the imaginary matrices. The number of columns and rows of the matrices must be specified.

The imaginary matrix has the same dimensions as the real one. Always returns them as a matrix, even if they have one row. Both matrices are returned in one list, and each matrix itself is a list of lists.

GetComplexMatrixOfSameSize(*matrixname, real_placeholder, imaginary_placeholder,* ***workspace** = 'base')*

Get both the real as the imaginary matrices. Instead of the dimensions, as with `GetComplexMatrix`, two matrices with the expected dimensions must be passed. Their values will not be changed. Always returns them as a matrix, even if they have one row. Both matrices are returned in one list, and each matrix itself is a list of lists.

PutComplexMatrix(*matrixname, real, imaginary,* ***workspace** = 'base')*

Put both the real as the imaginary matrices

The imaginary matrix has the same dimensions as the real one. Always returns them as a matrix, even if they have one row. Both matrices are returned in one list, and each matrix itself is a list of lists.

Execute

Execute any Matlab command.

MinimizeCommandWindow

Minimize the Matlab console window.

MaximizeCommandWindow

Maximize the Matlab console window.

SimplePlot(*fig, x_start, x_end, yvalues,* ***xlabel** = 'X',* ***ylabel** = 'Y',* ***title** = 'Simple plot',* ***window_title** = 'Matlab')***

Plot in one window the list of values.

The values are a tuple with one or more lists of values. Note that the number of x–axis positions must match the number of y–values. Specify `fig=-1` to get a new figure window.

Returns the number of the window.

8.2 INSTALLATION REQUIREMENTS

Matlab must be registered as an automation server before it can be used as an ActiveX object. Run **matlab /Regserver** to enable ActiveX usage.

8.3 PYTHON SCRIPT EXAMPLE

```
import matlab

m = matlab.Matlab()

m.SimplePlot(1, 1, 10, [[1,2,3,4,5,6,5,4,3,2]])

print m.Execute(’m2=[1,2,3; 4,5,6]’)
print ”get m2:”, m.GetMatrix(’m2’, 3, 2)

print ’imaginary 2x2’
print ”/                 \\”
print ”| 1 + 9i , 2 + 8i |”
print ”| 3 + 7i , 4 + 6i |”
print ”\\                 /”
m.PutImaginaryMatrix(’m3’, [[1,2],[3,4]], [[9,8],[7,6]])
print m.Execute(’m3’)
print m.GetImaginaryMatrix(’m3’, 2, 2)

del m
```

INDEX

INDEX