Concept EFB
User manual

840 USE 505 00 eng Version 2.6

33002260.00

Table of Contents

About the Book

At a Glance

Document Scope

This users manual should help you when generating and managing user defined functions and function blocks with Concept EFB.

Validity Note

This documentation is valid for Concept EFB Version 2.6 in Microsoft Windows 98, Microsoft Windows 2000, Microsoft Windows XP and Microsoft Windows NT 4.x. together with Borland C Compiler Version 5.01/5.02.

Note: Further current information can be found in the Info_EFB file in your Concept EFB installation.

Related Documents

Title of Documentation	Reference Number
Borland C 5.01/5.02 User Manual	
Concept User Manual	840 USE 503 00
Concept IEC Block Library	840 USE 504 00
Concept LL984 Block Library	840 USE 506 00

User Comments

We welcome your comments about this document. You can reach us by e-mail at TECHCOMM@modicon.com

Implementation in Concept EFB

1

Overview

Introduction

This chapter gives an overview of Concept_EFB.

What's in this Chapter?

This chapter contains the following topics:

Introduction

General

Concept EFB enables the user to generate user defined functions and function blocks.

User defined functions and function blocks can be managed using libraries (as can functions and function blocks delivered with Concept).

Elementary Function Blocks (EFBs

The elementary functions and function blocks included with delivery are organized into different libraries. The functions required can be selected by the user accordingly. These blocks are called EFBs (Elementary Function Blocks).

User Defined - EFBs (UDEFBs

The Concept EFB enables the experienced user to generate additional UDEFBs (User Defined - EFBs), and use them in their application. These functions and function blocks can be stored in a specific library by the user. This does not apply to libraries delivered with Concept.

Note: There is no difference between elementary and user defined functions/ function blocks within Concept. To simplify matters, EFBs will be described in general terms.

Languages

Concept applications can be created in several languages (FBD, LD, ST, IL) using the blocks provided according to IEC1131.

Programming language C

The creation of EFBs places other demands on the user than programming with IEC languages because blocks are developed in C. The programming language C is used because of its scope and functionalities.

The function range covered by C, is reduced for the requirements here. This reduction is explained in more detail in following chapters.

Integrated development environment

Concept EFB provides a complete development environment, for writing, creating and installing libraries. Tasks are started according to the selection of menu points in the development environment. This calls batch processes which carry out all required steps automatically. This means that the Borland C-Compilers or Turbo Assemblers do not need to be called manually.

Restrictions

General

It is **not allowed** within the EFB C-code to write on inputs or to read from outputs.

Note: This problem only concerns boolean datatypes.

Write on inputs

It is **not allowed** to write on inputs inside an EFB, although Concept 2.1 generates no errors if you do this. This can cause problems when the user connects a write-protected literal value to the pin.

However from Concept 2.2, Concept intercepts a write on boolean inputs as soon as they are connected to 0x or 1x registers (mapping them to variables causes no problems).

With Concept 2.5 and higher you are allowed to write to EFB inputs if the pin type `Var_InOut` is used. The pin type `Var_InOut` does not allow the user to attach literals or booleans to the pin. (For the description of the new pin type definition `Var_InOut` refer to chapter *Var_InOut, p. 60*).

Read from outputs

It is **not allowed** to read from outputs inside an EFB, although Concept 2.1 generates no errors if you do this. This can cause problems when the user connects a located variable to the pin.

Advantages of User Defined EFBs

Independent Libraries

By using the language C, it is possible for experienced application programmers to create simple to complex functions which can be used many times in different applications. The functions are saved in libraries independent of the respective application and are generally available for all applications after installation.

Protected Function Blocks

The technology and structure of the function blocks is not transparent for others because they are black boxes.

Improvements to Runtime and Memory Requirements

The function blocks created with Concept EFB can be used in all of Concept's IEC languages. Compared with equivalent algorithms based on multiple use of primitive functions/function blocks, the use of UDEFBs represents a great improvement regarding runtime and memory requirements.

New functions of Concept EFB 2.5 and higher compared to Concept EFB 2.2

General

Concept EFB 2.5 and higher contains the following new functions and modifications compared to Concept EFB 2.2.:

- New IEC PIN type `Var_InOut`.
- Creation of an installation diskette for distributing libraries.
- Reduction of the development path name to 22 characters.

Var_InOut

The new PIN type `Var_InOut` enables the user to generate EFBs containing PINs, that can be used simultaneously as the input and output of a block.

See chapter *Keywords for Input and Output PINs, p. 58*,*Var_InOut*.

Installation diskette

Libraries are distributed to computers by creating an installation diskette on the system with ConceptEFB installed.

See chapter*Installation of Libraries on Different Computers, p. 66*.

Development path name

The length of the development directory path name is checked and is limited to 22 characters.

See information in chapter *Directory Structure, p. 72*.

Installation Instructions for Concept EFB

2

Overview

Introduction

This chapter gives an overview of the installion of Concept EFB.

What's in this Chapter?

This chapter contains the following topics:

Introduction

General

Concept EFB is an independent tool. Since both Concept and Concept EFB use common components they can not run in parallel.

Required components

The following components are required for installation:

- Concept
- Concept EFB
- Borland C 5.01/5.02
- 16 bit and 32 bit Turbo Assembler.

Sequence

When installing Concept EFB you must observe the installation sequence for installing the individual components. The list of components above shows a useful sequence

Also see chapter *Installation Order, p. 15*.

Installation Order

General

When installing Concept EFB, certain components must be installed in a designated order.

Concept must be installed before Concept EFB.

The development package Borland C 5.01/5.02 must be installed before the Turbo Assembler.

Borland C 5.01/5.02 and Turbo Assembler can be installed before or after Concept.

The following order for required components is also a sensible installation order.

Concept

The installation of Concept must take place before the installation of Concept EFB, because the tool is installed in the same directory as the Concept application.

Concept EFB

The installation of Concept EFB can only take place after the installation of Concept.

Borland C 5.01/ 5.02

To develop EFBs, it is necessary to also install the correct development package Borland C 5.01/5.02 on the selected computer. This can take place before or after the installation of Concept EFB, but must be completed before function blocks are created with Concept EFB.

16-bit and 32-bit Turbo Assembler

Additionally, both a 16-bit and a 32-bit variant of the Turbo Assemblers are required (`TASM.EXE` and `TASM32.EXE`).

Installation of Concept EFB

Versions

Note: When installing Concept EFB is it important to note that only the same versions of Concept and Concept EFB can be installed together.

Installation of the CD with autorun

The following section describes the installation of the Concept EFB CD when the CD drive autorun function is switched on:

Step	Action
1	Place the CD ROM in the CD drive.
2	The **Setup** program is usually started automatically by the CD after a short time.
3	Follow the onscreen instructions.

Installation of the CD without autorun

If the Setup program is not started automatically it means the CD drive autorun function is deactivated. In this instance do the following:

Step	Action
1	Open the My Computer icon on the desktop.
2	Select CD drive and open it with a double-click.
3	Start the **Setup** program displayed in the directory with a double-click.
4	Follow the onscreen instructions. **Result:** Concept EFB is installed in the existing Concept program group

Installation variations

Note: These installation instructions only cover one method of installation. Installation can be carried out using the My Computer route and also clicking the cursor on **Execute**, and additionally, instead of double-clicking to open the file, you can simply use the mouse or keyboard and open the file using the **Enter** key. Also see the operating instructions for Windows.

Installation of Borland C 5.01/5.02

General

Borland C 5.01/5.02 is a development package that contains the compiler, tools and service programs for creating EFB libraries.

Installation

The installation of Borland C 5.01/5.02 is carried out according to the instructions provided in the development package.

Note: The Borland C compiler version 5.5 currently freely available on the internet is not compatible with Concept EFB and only generates 32 bit code.

Installation the 16 bit and 32 bit Turbo Assembler (TASM and TASM32)

General

In addition to the Borland C 5.01/5.02 development package it is necessary to install the assembler programs TASM.exe and TASM32.exe.

You are adivised that blocks created in C can only be compiled using an assembler temporary file because of special requirements, and cannot be copied directly from the available IDE (Integrated Developers Environment).

Installation sequence

Installing the program files (TASM.exe and TASM32.exe) can only be carried out if the development package Borland C 5.01/5.02 has already been installed.

Installation directory

For code generation (16 bit / 32 bit) it is necessary to copy the files (`TASM.EXE` and `TASM32.EXE`) to the Borland C compiler´s bin directory.

Included with delivery

The Turbo assembler files are not included with the delivery of the Borland C 5.01/ 5.02 develpoment package, and must be purchased additionally.

Note: TASM32.EXE is included in the C Builder 5.0 professional package from Inprise (also includes Borland Compiler 5.02)

System Settings

General

After installing all required programs the following information must be entered in the system (Win95/98 or WinNT):

- the BIN directory (Borland C) and
- the Include directory (Borland C).

Set system variable PATH

Enter in `Autoexec.bat` Win95/98 i

- `Path = C:\BC5\BIN; C:\BC5\INCLUDE; %PATH%`

or for WinNT in the **Settings**→ **Control Panel** for the variable PATH

- `C:\BC5\BIN; C:\BC5\INCLUDE;`

Setting the Options in Concept EFB

General

After the installation of all necessary programs, the following points must be defined in Concept EFB:

- the development directory (EFB development directory),
- the include directory (Borland C) and
- the desired hardware platform (16-Bit / 32-Bit).

Starting Concept EFB

Start Concept EFB using **Start** → **Programs** → **Concept** → **Concept EFB**.

The Concept EFB program window is shown on the screen.

Using the menu item **Options**, you can make all of the following definitions.

Create development directory

After starting Concept EFB the first time, a development directory (EFB development directory) must be created. When developing function block libraries, the source code and the generated files are placed in this directory and its subdirectories before they are installed in Concept.

Using **Options** → **Create Development Path...**, you invoke a dialog box where you can create the path used to store all libraries created with Concept EFB. Also see Chapter *Options Menu, p. 33*.

The development path is shown in the title bar of the program window.

Note: The path name of the Development directory is limited to a maximum of 22 characters. Therefore, placing this directory in the root directory of a drive is recommended. If an attempt is made to enter a path with more than 22 characters for the Development directory, an error message is given.

Select development directory

If a development path already exists, you can use **Options** → **Set Development Path...** to invoke a dialog box where you can select the path for the development directory. Also see Chapter *Options Menu, p. 33*.

Select Borland include directory

The path to the include directory for the Borland C compiler also has to be entered in Concept EFB.

Using **Options** → **Set Borland Include Path...**, you can invoke a dialog box where you can enter the path to the include directory (e.g. `C:\BC5\`). Also see Chapter *Options Menu, p. 33*.

Select hardware platform

The hardware platform, for which the function block will be developed, must be entered in **Options**.

Using **Options** → **Options...**, you can invoke a dialog box where you can set the options for the generation of function block libraries.

Here, you have the possibility to select whether code generation should be for 16-bit and/or 32-bit.

You can also set up whether information for the debugger should be generated when creating the function blocks. This option is necessary if you wish to use the Borland debuggers TDW.EXE and TD32.EXE.

You may also determine if log files generated when creating libraries should be automatically shown on the screen after generation. We recommend setting this option.

Also see Chapter *Selecting the Hardware Platform, p. 50*.

Checking the Installation

General

After all required programs have been installed, the system settings configured (see chapter *System Settings, p. 18*) and the required options set in Concept EFB (see chapter *Setting the Options in Concept EFB, p. 19*), the SAMPLE library delivered with Concept EFB can be used to run a check for successful installation. This tests the interaction of the installed applications.

Start Concept EFB

Start Concept EFB via **Start** → **Programs** → **Concept** → **Concept EFB** if it hasn't already been started.

The Concept EFB program window appears onscreen.

Select the SAMPLE library

Open a dialog window using **File** → **Open Library** in which you can select an existing library.

A library contained in the development path can be selected from the list box given.

Mark the SAMPLE example library and close the dialog box by clicking **OK**.

The marked library is selected for editing and is displayed in the program window title bar.

If all components are correctly installed, the blocks available in the SAMPLE library can now be seen in the **Objects** → **Select EFB** dialog box.

See also chapter *File menu, p. 27*.

Note: If the **File** → **Open Library** menu is inactive or the development path is missing, this is indicated in the Concept EFB window title bar. (see chapter *System Settings, p. 18*).

If the message window shows **No path to Borland Compiler bcc.exe found**, check the correct entry for the Borland C compiler path. (see chapter *System Settings, p. 18*). Paths are only read when restarting the computer with a 16 bit development environment. Restart your computer so the path can be read.

Generate Files

Start generation of all necessary files using **Library** → **Generate Files** (Prototype and source files) for an EFB library.

The files required for the conversion are generated for all EFBs in this library (SAMPLE) and this procedure is logged in a DOS window.

After generation, an editor window `log.txt` is shown with the logged messages.

See also chapter *Library Menu, p. 29*.

Note: The editor window (`log.txt`) is only shown automatically if in **Options** → **Options...** the item **Show Logfile** is selected. See Chapter *Options Menu, p. 33*.

Otherwise the logfile can be opened using **Library** → **Logfile**.

Make

After using **Generate Files** to generate the necessary files for the library, make the library using **Library** → **Make**. Making the library is logged in a DOS window. After creation, an editor window `log.txt` is shown with the logged messages.

See also chapter *Menu Library*.

Error log

If errors occur when compiling the example library, a message window is shown with tips for the user.

After the message is acknowledged, an editor is shown (`log.txt`) with the logged messages (as when successful).

Note: Problems such as the message **Unable to open include file** are mostly caused by the installation of the Borland C compiler. In this case, check if the path is correct.

Incorrect Versions

If problems occur when compiling the files (e.g. message `Error: pro failed with -314`, this is mostly caused by an faulty Borland C compiler installation or the version is incompatible.

In this case, check the available program versions of the files belonging to the compiler using the following table.

File name	Program Version for BC5.01	Program Version for BC5.02
Tlink	7.1.30.1	7.1.32.2
Tlink32	1.6.71	2.0.68
Make	4.0	5.0
Bcc	5.0	5.2
Bcc32	5.0	5.2

The check is best made by opening a DOS window and entering the respective program names at the command line.

Concept EFB Main menu

3

Overview

Introduction

This chapter gives an overview of the main commands from the main menu of Concept EFB.

Menu commands enable the selection, generation and management of function blocks in libraries.

What's in this Chapter?

This chapter contains the following topics:

Introduction

General

The Concept EFB program window comprises:

- the title bar displaying the development path,
- the menu bar with the drop down menus **File**, **Library**, **Objects**, **Options**, **Help**,
- the toolbar giving direct access to individual submenus,
- and the main window.

Prerequisites

In order to select, generate and manage functions or function blocks with Concept EFB, all required system settings must first be carried out (see chapter *System Settings, p. 18*) and all options set in Concept EFB (see chapter *Setting the Options in Concept EFB, p. 19*).

File menu

General

The **File** menu offers the following possibilities:

- Creation of libraries,
- Selecting libraries,
- Conversion of libraries,
- Import function blocks into libraries,
- Move function blocks between libraries and
- end the program

As long as no library has been selected, two of the submenus in the **File** menu (**Import EFBs...** and **Move EFBs...**) are shown in grey. As soon as a block library is selected the submenus are activated.

New Library

The submenu **New Library** opens a dialog box for creating a new block library in the development path specified. When the dialog box is ended with **OK**, an empty library is created and is immediately selected for editing.

The submenu **New Library** can also be reached via a button in the toolbar.

Open Library

The submenu **Open Library** opens a dialog box for selecting an existing library. A list box offers the user the block libraries available for selection in the development path. When the dialog box is ended by clicking **OK**, the library marked in the list box is selected for editing and the dialog box is closed.

The submenu **Open Library** can also be reached via a button in the toolbar.

Convert Library

The submenu **Convert Library** opens a dialog box for entering the path to libraries that were created using an older version of the Concept EFB. When a library is selected and the dialog box ended with **OK**, the libraries found in this path and all the EFBs they contain are converted for the new Concept EFB version and saved in the current development path.

Import EFBs

The submenu **Import EFBs** opens a dialog box for entering the path of the library to be imported. All EFBs available in this library are imported into the active library when the dialog box is ended with **OK**. The imported blocks are then contained in both libraries.

Move EFB

The submenu **Move EFB** is similar to the submenu described above **Import EFBs**. The difference is that after entering the library path a further dialog box appears where the user can select the name of an EFB. Only this block is imported into the library when the dialog box is ended with **OK**. The imported block is then deleted from the source library.

Note: The submenus **Convert Library**, **Import EFBs** and **Move EFB** setup the standard file structure of a library or EFB. User header files are ignored and must be copied manually. See also chapter *User Includes, p. 76*.

Print

The submenu **Print** opens a dialog box where you can select whether the information for a library should be printed or whether EFBs available in the library should be printed.

Exit (Alt+F4)

The submenu **Exit** ends the program.

Library Menu

General

The **Library** menu offers the following possibilities:

- definition of derived data types,
- display derived data types as type definitions in the C programming language,
- generate all files required for compiling the EFBs,
- creating a library,
- recompile all files belonging to a library,
- install a library in Concept,
- display the last protocol file created,
- display a summary description for all EFBs in the active library,
- search for available backup files as required and
- delete the active library.

As long as no library is selected, only the submenu **Logfile** is active and the user can only see the most recently created protocol file. The submenus become active once a block library has been selected

Derived datatypes

The **Derived datatypes** submenu opens an editor window where the user can define derived datatypes. The syntax for describing data types is given by IEC 1131-3. The submenu **Derived datatypes** can also be reached via a button in the toolbar.

C Header of derived datatypes

The submenu **C Header of derived datatypes** activates an editor window where data types created in**Derived datatypes** are displayed as type definitions in the C programming language. This list can only be seen after the function **Generate Files** was invoked.

Generate Files

The submenu **Generate Files** generates all files required for compiling. Files are only created for the EFBs within a library that have a modified or new definition file.

The submenu **Generate Files** can also be reached via a button in the toolbar.

Make

The submenu **Make** invokes the C compiler and generates the compiler object files (.obj) for the EFBs in the active library.

The submenu **Make** can also be reached via a button in the toolbar.

Build

The submenu **Build** creates a library in a similar way to the **Make** function. In comparison to the commands described above, no dependencies of the individual files to each other is taken into account. All files belonging to the library are recompiled.

Install

The submenu **Install** is used to install the library created by **Make** or **Build** in Concept. A library can be used by Concept as soon as it has been installed.

The submenu **Install** can also be reached via a button in the toolbar.

Logfile

The submenu **Logfile** displays the protocol file created by the last compile/generate files.

Summary

The submenu **Summary** opens an editor window which display a summary of all EFB descriptions belonging to this library.

Find backups

The submenu **Find backups** searches for any backup files in the available library EFB. These backups are created during edit sessions or by the function **Generate Files** and can be used to restore files using cut and paste.

Delete library

The submenu **Delete library** deletes the active library after the security message is confirmed. Concept EFB is then in the same status as after a program start and waits for the user to select another library.

Objects Menu

General

The **Objects** menu offers the following possibilities:

- a framework for creating a new function,
- a framework for creating a new function block,
- selecting an existing EFB for editing,
- edit the Definition file of the active EFB,
- edit the Source file of the active EFB,
- display the Prototype file of the active EFB,
- select a backup file for the active EFB,
- delete the backup files of the active EFB,
- delete all files and directories of the active EFB,
- deactive an EFB and
- reactive an EFB.

As long as no block library has been selected, the **Objects** menu only contains inactive submenus. Once a block library has been selected, the submenus **New Function** , **New Function Block**, **Select EFB** and **Reactivate EFB** can be used.

All submenus become available as soon as an EFB has been created using **New Function** or **New Function Block** or an EFB has been selected using **Select EFB**.

New Function

The framework for a new function is created using the submenu **New Function**, once the user has entered the author´s name and name of the EFB in a dialog box. After creating the framework the definition file is automatically displayed in an editor window.

The submenu **New Function** can also be reached via a button in the toolbar.

New Function Block

The framework for a new function block is created using the submenu **New Function Block**, once the user has entered the author´s name and name of the EFB in a dialog box. After creating the framework the definition file is automatically displayed in an editor window.

The submenu **New Function Block** can also be reached via a button in the toolbar.

Select EFB

The submenu **Select EFB** opens a dialog box for selecting an EFB. The user can select the EFBs available in the library from a list box. When the dialog box is ended by clicking **OK**, the EFB marked in the list box is activated (selected for editing) and the dialog box is closed.

The submenu **Select EFB** can also be reached via a button in the toolbar.

Definition

The submenu **Definition** opens an editor window which displays the Definition file (*.fb) for the active EFB and allows it to be edited.

The submenu **Definition** can also be reached via a button in the toolbar.

Source

The submenu **Source** opens an editor window which displays the Source file (*.fb) for the active EFB and allows it to be edited.

However, after creating a new EFB, the submenu **Generate Files** must first be called, from the **Library** menu, since the source file is created using the definition file.

The submenu **Source** can also be reached via a button in the toolbar.

Prototype

The submenu **Prototype** opens an editor window which displays the Prototype file (*.h) of the active EFB. The prototype file is generated using **Generate files**. The file may be edited, however, all changes made after calling **Generate files** are lost as the prototype file is recreated. This means this function is only used for displaying the created prototype file.

Backup

The submenu **Backup** opens a dialog which allows you to recover an old source file of the active EFB.

Delete backups

The submenu **Delete backups** deletes the backup files available for the active EFB.

Delete EFB

The submenu **Delete EFB** deletes all files and directories belonging to the active EFB after confirmation is given to the security message. Concept EFB is then in the same status as after selecting a library and waits for the user to select another EFB.

Deactivate EFB

The submenu **Deactivate EFB** deactivates the EFBs of a library. Concept EFB is then in the same status as after selecting a library and waits for the user to select another EFB.

Reactivate EFB

The submenu **Reactivate EFB** opens a dialog box where you can reactivate a currently deactivated EFB. The reactivated EFB then becomes the active (i.e. currently selected) EFB

This submenu is the counter part to the **Deactivate EFB** function.

Options Menu

General

The **Options** menu offers the following possibilities:

- select a development directory,
- create a new development directory,
- assign the Include directory of the Borland C compiler,
- select the hardware platform (16 bit and/or 32 bit),
- specify whether information should be generated for the debugger and
- specify if the log file should appear automatically.

Set Development Path

The submenu **Set Development Path** opens a dialog for selecting the development directory where all libraries created with Concept EFB should be stored (or are).

Create Development Path

The submenu **Create Development Path** opens a dialog for creating a new development directory where all libraries created with Concept EFB should be stored.

Set Borland Include Path

The submenu **Set Borland Include Path** opens a dialog box for declaring the Include directory of the Borland C compiler.

Note: The Include directory must be specified before creating a library for the first time, otherwise Concept EFB cannot find the Standard Borland Include files.

Options

The submenu **Options** opens a dialog box in which options for generating EFB libraries can be set. The user has the opportunity here to select the hardware platform and therefore determine whether the code generation for 16 bit and/or 32 bit should be allowed.

Furthermore, the user can set here whether or not information for the debugger should be generated when creating the block.

A further point determines whether the resulting log file should automatically be shown on screen after creating and generating a library.

Help Menu

General

The **Help** menu offers the following possibilities:

- To open the online help for Concept EFB.
- To display version information about Concept EFB.

Contents

The online help provided for Concept EFB can be opened using the submenu **Contents**. This is available after Concept EFB is installed in the Concept install directory.

About

Version information about the installed Concept EFB can be displayed using the submenu **About**.

Creating and Editing EFBs

4

Overview

Introduction

This chapter gives an overview of creating and editing function blocks.

What's in this Chapter?

This chapter contains the following topics:

Creating EFBs

General

An EFB (Elementary Function Block), which can be defined as a function or a function block (see chapter*Differences between functions and function blocks, p. 51*), must always be a component of an EFB library.

All functions and function blocks available in a library are compiled together using the Borland-C compiler and are available for use after the EFB library is installed in Concept. The only exceptions are the EFBs of a library that have been deactivated.

Prerequisites

In order to create functions or function blocks with Concept EFB, all required system settings must be made (see chapter *System Settings, p. 18*), and all options set in Concept EFB (see chapter *Setting the Options in Concept EFB, p. 19*).

Development directory

As described in chapter *Setting the Options in Concept EFB, p. 19*, a development directory must be created.

Several libraries can be created within a development directory. However, only one version of a library may exist in a development directory.

Note: During installation of the EFB library in Concept, EFB creation and library creation, names entered are checked against the Concept database. If the names alredy exist an error is generated. Names of libraries and EFBs must be unique.

It is possible to work with different development directories which contain libraries which are at different stages of development.

You can switch between the different directories using **Options** → **Set Development Path**. This opens a dialog window where you can select an existing directory as a development directory. With this selection, clicking the **OK** buttons checks whether the directory selected is a valid directory for creating EFB libraries before switching directories. If the directory entered does not exist or does not contain the files required, the directory switch is interrupted with an error message. Otherwise the directory enterd is selected. The active development directory is displayed in the applications title bar until a library stored in the directory is opened.

Procedure

The creation of EFBs is done in several steps. After opening an EFB library, follow the procedure in the table below when creating an EFB:

Step	Action	Result/ Remark
1	Use **New Function** to create a new function or create a new function block using **New Function Block**. See chapter*Objects Menu, p. 31*.	The definition file is created and displayed in an editor window.
2	Edit the Definition file (*.fb) as described in the chapter *Editing Definition files, p. 38*.	
3	After editing, create all required files using **Generate Files**. See chapter*Library Menu, p. 29*.	The source file (*.c) and pototype file (*.h) are created.
4	Edit the source code file (*.c) as described in the chapter *Editing a source file, p. 40*.	This file contains the actual programming for the block.

Testing EFBs

To test created EFBs see chapter *Testing Created EFB, p. 67*.

Editing Definition files

General

When creating a new EFB (see *Creating EFBs, p. 36*) the dummy Definition file created only shows a framework for the actual block. The file should be edited according to the requirements on the block.

The definition file contains general information about the EFB as well as information about all the EFB pins. It defines the functional and graphical EFB interface to Concept.

Note: After making changes in the definition file you must run **Generate Files** for these changes to become active. **Generate Files** also creates a new dummy source file. If you require that the old source file be retained then you must open it with an editor and save it. This save updates the time stamp on the source file and indicates that the changes in the definition file have already been moved across; a new source file is not created. This means however that you may (depending on your changes in the definition file) have to make the changes in your source file manually. If you do not save the source file it will be copied to a backup file and a new dummy source file is created.

Open the definition file

Open the definition file using **Objects** → **Definition**.

Example of a definition file

Definition file for the example block **MY_FUBL1**.

```
//:-----------------------------------------------------------
//:SUBSYSTEM:   EFB - Elementary Function Block
//:
//:MODULE:            ..\EFBTEST\MY_FUBL1.FB
//:
//:------------------------------------------------------------
//:Revision: 2.1  Modtime: 22 Jan 1998  AUTHOR: Peter Alig
//:------------------------------------------------------------

Declaration of Elementary Function Block: MY_FUBL1
Author: Peter Alig
Editor Group: Test Group
Major Version: 1
Minor Version: 0
Description: A full working test example.
//
// Not sure what to do ?
// Try generate-files, make and install on this working
example!
//
// Example:
// --------
Rising Edge Detector: BOOL CLK # count up clock
Input:        INT  INCVAL      # increment value per clock
Input:        INT  MAXCOUNT    # limiter on upcount
State Output: INT  COUNTOUT    # counter output value
Output:       INT  TICKS    # ticks since last change of INCVAL

// use the next line to declare an internal state structure
Internal State: BOOL initDone  # initialising done on first run
// See the dummy source file after doing ’Generate Files’ for
more information
```

Key word

The definition file is comprised of different key words and commentary. The source file and prototype file are created using these key words when generating the files (see chapter*Creating EFBs, p. 36*).

Key words are terms reserved for special requirements that may not be used as normal terms

Note: Upper and lower case text is differentiated between when evaluating the definition file, so that `Sum`, `sum` and `SUM` are treated as different terms.

A list of permitted key words, their meanings and syntax descriptions can be found in chapter *Keywords of a definition file, p. 46* and *PIN Syntax, p. 47*

Editing a source file

General

After the source and prototype files have been created with the Definition file using **Generate Files** (see *Creating EFBs, p. 36*), the source file must be edited, since after the automatic generation this only comprises of a dummy function definition.

Note: After each change made to the definition file the source file must be updated automatically using the menu command **Generate Files**. The old version of the source file is saved by the system as a backup file.

Open the source file

Open the source file using **Objects** → **Source**.

Example of a dummy source file

Source file for the example block **MY_FUBL1**.

```
//:-----------------------------------------------------------
//:SUBSYSTEM:   EFB - Elementary Function Block
//:
//:MODULE:      MY_FUBL1
//:
//:------------------------------------------------------------
//:Revision: 1.0  Modtime: 30 May 2000  AUTHOR: Peter Alig
//:------------------------------------------------------------
//:DESCRIPTION: A full working test example.
//:
//:REMARKS:
//:------------------------------------------------------------
#include "EFBTEST.I"

extern"C" BOOL FB_CALL_CONV MY_FUBL1(
   const PTR_BOOL     CLK      ,   //count up clock
   const PTR_INT      INCVAL   ,   //increment value per clock
   const PTR_INT      MAXCOUNT ,  //limiter on upcount
         PTR_Istat_MY_FUBL1 Istate ,  // internal state
         PTR_INT      COUNTOUT ,    //counter output value
         PTR_INT      TICKS    )   //ticks since last
                                     change of INCVAL
{
if_FALSE(Istate->initDone)
  {
  // initialize first time
  Istate->initDone = TRUE;
  }

AliPutFbdError(E_DUMMY_SOURCE_CODE);
return FALSE;
}
```

Command in C-Syntax

You can now add the required commands in C syntax. Also see chapter *Code Limitations, p. 45*.

Floating point literals

Floating point literals are not allowed in EFB code, because of the Assembler code generated in Borland C. This assumes that a DGROUP segment is available which is not available in the PLC.

In order to use a floating point literal it is necessary to define a variable and initialize it with the constant value
required

```
REAL   k0_77=  0.77;
```

Also see chapter *Recognizing Instructions with DGROUP Segment, p. 91*.

Backup the Source file

General

After each change made to the Definition file, **Generate Files** must be used to automatically update the source file.

This also saves the old source file as a backup file (backup00.c, backup01.c ...).

Time stamp

Concept EFB uses the files time stamp as the criteria to determine which files must be backed up.

Editing the prototype file

General

The prototype file can be edited, however, all changes made after calling **Generate files** are lost since the prototype file is recreated. This means this function is only used for displaying the prototype file created.

Open the prototype file

Open the prototype file using **Objects** → **Prototype**.

Example of a prototype file

Prototype file for the example block **MY_FUBL1**.

```
// Lines below are generated by tool, all manual changes will
be overwritten.//
//:------------------------------------------------------------
//:SUBSYSTEM:   EFB - Elementary Function Block
//:
//:MODULE:      ..\EFBTEST\MY_FUBL1\MY_FUBL1.H
//:
//:------------------------------------------------------------
//:Revision: 1.0  Modtime: 30 May 2000  AUTHOR: Peter Alig
//:------------------------------------------------------------
//:DESCRIPTION: A full working test example.
//:
//:REMARKS:
//:------------------------------------------------------------

// Elementary Function Block:  MY_FUBL1
//
//                        ____________________
//                       |                    |
//                       |      MY_FUBL1      |
//                       |                    |
//                BOOL ---> CLK               |
//               INT ---| INCVAL    COUNTOUT |--- INT
//               INT ---| MAXCOUNT     TICKS |--- INT
//                       |____________________|
typedef struct Istat_MY_FUBL1 {
      BOOL       CLK_old;   // internal state
      BOOL       initDone;  //initialising done on first run
} Istat_MY_FUBL1, FB_PTR_CONV *PTR_Istat_MY_FUBL1;



//:FUNCTION:------------------------------------------------
extern "C"
                              //
                              // R,   Correctness of execution
BOOL FB_CALL_CONV MY_FUBL1
(const PTR_BOOL  /*CLK    */,  // I, count up clock
 const PTR_INT   /*INCVAL */, // I, increment value per clock
 const PTR_INT   /*MAXCOUNT*/,// I, limiter on upcount
       PTR_Istat_MY_FUBL1 /*Istate */, // I/O, Internal State
       PTR_INT   /*COUNTOUT*/,// I/O, counter output value
       PTR_INT   /*TICKS */ );// O, ticks since last change
//:------------------------------------------------------------
//:DESCRIPTION: A full working test example.
//:
//:REMARKS:
//:------------------------------------------------------------
```

Graphical overview

The prototype file defines the graphical overview of the defined inputs and outputs.

The inputs are always shown on the left and the outputs always on the right of the EFB frame.

The positions of the outputs corresponds to the positions defined when editing the definition file.

EFB name

The name of the function/function blocks i.e. the function block type is displayed in the centre of the EFB frame.

Construction of an EFB

General

The code used in an EFB has specific limitations. Also see chapter *Code Limitations, p. 45*.

Configuration section/ Runtime range

It is advisable to seperate your EFB code into two sections, a standard running code for normal scans and initialising code which will be used to initialise the EFB´s internal data (should it have any). The initialising code should only be called during initialisation of the PLC otherwise the normal code will be used (`if ... then ... else`). See the example in chapter *Editing a source file, p. 40*.

Runtime variations

The EFB should be written as far as possible so that the runtime remains the same for each call. Otherwise it could lead to unpredictable variations in PLC cycle times.

Longer block tasks should be distributed among PLC several cycles where necessary in order to minimize runtime variations.

Code Limitations

General

The creation of EFBs with Concept EFB is different in some repects to creating a "normal" C function. This chapter describes the differences.

Source file

EFBs have the following limitations:

- API functions can only be called in the main function of EFB,
- the number of in/out parameters on an EFB (not including `Internal State`) is limited to 32 respectively,
- the maximum size of a variable is limited to 64 kbytes,
- local variables can be included but the maximum stack size is 256 bytes,
- local variables are not initialized by Concept - this must be done within the EFB.

Code generation

A PLC operating system is different to a normal PC.

For example, no DGROUP data segments exist in PLC programs.

The following limitations should therefore be observed regarding code generation with the Borland C compiler:

- Static variables may not be used,
- Sub-functions that are called from within an EFB must be defined as static,
- Floating point operations must be handled with care as it has been shown that data segments are also used for saving values for example, floating point literals may not be declared (see chapter *DGROUP Segments*),
- the code generated by the Borland C compiler must be supported by the PLC operating system. This is guaranteed for standard instructions in C, but not for all functions of the standard C library. Generally, string operations or input/output functions and storage creation functions are not allowed.

Unique names

All names used for EFBs must be unique. This does not only apply within a block library, but, also for later uses of blocks in Concept generally because many different libraries can be used simultaneously in Concept.

Names of libraries, blocks and data structures must be unique.

Keywords of a definition file

Keyword

The following table specifies permitted keywords allowed in definition files and gives a brief description of the individual terms.

Examples of keywords can be seen in the SAMPLE example library.

Keywords with descriptions:

Permitted keywords	**Description**
`//`	Line coments, all characters after these characters are treated as commentary
`Declaration of Elementary Function Block`	This keyword describes the declaration of a function block
`Declaration of Elementary Function`	This keyword describes the declaration of a simple function
`Description`	A block is described as follows. See 1).
`Remarks`	Remarks follow. See 1).
`Special header information`	Defines header information which will be copied to the EFB header file and must be C compatible. See 1).
(empty space)	Subsequent lines in a multiple line information block in this file are identified by a leading empty character. See 1).
`Editor group`	The group to which this block belongs
`Major version`	Version number of the EFB (major)
`Minor version`	Version number of the EFB (minor)
`Input`	Declaration of an input pin. See 2) + 3).
`State output`	This creates an output pin which retains its value. Normally used for booleans.
`Rising Edge Detector`	Declaration of an input for a signal that should be recognized as a rising edge. See 2).
`Falling Edge Detector`	Declaration of an input for a signal that should be recognized as a falling edge. See 2).
`Output`	Declaration of an output pin. Siehe 2).
`Internal state`	Declaration of a concealed data structure input. See 4) + 6).
`Var_InOut`	Declaration of a PIN that can be used as both an input and an output. See 5).

Permitted keywords	Description
1 Several lines of information can follow these keywords. This information is limited by another keyword or by the end of the file. Subsequent lines must be preceded with a space (i.e. white space).	
2 The syntax for an input or output pin is explained in the chapter PIN syntax.	
3 Declarations for input pins may be expandable (see chapter PIN syntax).	
4 These keywords create a concealed parameter "Internal state", which can be used to save internal block information outside the EFB.	
5 When the keyword is entered a PIN is created which is displayed on both the input side and output side (directly opposite) of the block.	
6 Only possible in function blocks.	

PIN Syntax

Input / Output PINs

The inputs or output PINs of an EFB should be declared according to the following scheme:

```
<key>[(<exp_info>)]:[=|.|+] <typ> <name> [#<comment>]
```

The following table lists the parameters for declaring PINS:

Parameter	Meaning
`<key>`	Keyword from the table *Keyword, p. 46*
`<exp_info>`	Additional information for `extendible PINs` (see below)
`<typ>`	PIN data type
`<name>`	PIN name
`<comment>`	Commentary, automatically cut off at the end of the line
`[=|.|+]`	optional entry for outputs (see below)

Extendible PINs

The keyword `Input` for input parameter can be expanded as follows for `extendible PINs`:

```
<min>..<max>,default=<def>
```

The following table lists the parameters for expansion:

Parameter	Meaning
`<min>`	Minimum number of inputs to declare
`<max>`	Maximum number of possible inputs
`<def>`	Standard number of inputs available

Position of PINs

The graphical alignment of the PIN on the side of the EFB can be influenced using this declaration.

Parameter	Meaning
=	the output is defined with the same data type as is available at the input at the same hight on the other side (default)
.	it is not searched for according to an input of the same data type
+	this symbol can be entered multiple times. Each symbol entered jumps one position in front of the output symbol

Note: The position of `Var_InOut` pins can not be influenced. `Var_InOut` pins will always appear at the top position.

Concealed PIN name

The input and output labels are displayed in Concept as standard. These can be concealed using the `hide(<variable name>)` syntax in the definition file (*.fb).

Example:

`Input:    BOOL  in1` #Input of datatype BOOL with name in1 at position 1

`Input  :++  REAL  in2` #Input of datatype REAL with name in2 at position 4, the two pin positions above are skipped.

`Output :=  REAL  out1` #Output of datatype REAL at the same position (4) as a REAL datatype on the input side

`Output :=   REAL  hide(out2)` #Output of datatype REAL. The PIN-name out2 will not be displayed on the EFB.

EN / ENO

In the Concept graphical languages (LD, optional in FBD), enable input PINs (**EN**) or equivalent output PINs (**ENO**) are available.

The management of the **EN** PIN is carried out completely by the system. If the value of **EN** is set to FALSE in the application, the C code of the EFB is not carried out.

The value of the **ENO** PIN coressponds to the return value of C function of the EFB.

Note: If error handling occurs in EFB (e.g. false EFB parameter in the application) the return value should be FALSE.

Background Information for EFB Design

5

Overview

Introduction

This chapter provides background information for designing EFBs.

What's in this Chapter?

This chapter contains the following topics:

Selecting the Hardware Platform

General

The hardware platform that the function block will be developed for is entered under the menu item **Options**.

16-bit / 32-bit

For PLCs working with an Intel 80186 or 80286 CPU, 16-bit code must be created.

For PLCs equipped with an 80386 or higher, 32-bit code is needed. (exception: 140 CPU 424x0 executes 16-bit code).

Concept itself selects the appropriate type of code at download according to which hardware is configured.

The following table shows the assignments of the platforms to the PLCs supported by Concept.

PLC	16 Bit Platform (Dos16)	32 Bit Platform (Win32)
Quantum	140CPU x13x0, 140CPU 424x0	140CPU x341x
Momentum	All	-
Compact	-	All
Atrium	-	All

Setting

Using **Options** → **Options..** , you can get to a dialog box where you can set the options for the generation of function block libraries.

Here, you have the possibility to select if code generation should take place for 16-bit and/or 32-bit.

Testing

If both 16-bit and also 32-bit code should be created, both versions must be tested separately.

Normally there should be no difference between both versions, but under certain circumstances it has been observed that particular blocks work perfectly on a certain platform, however cause errors on others.

This is due to the fact that two different compilers are used for different platforms.

You may find that you need different code in the 16 bit and 32 bit versions of an EFB for it to run correctly on all platforms.

Differences between functions and function blocks

General

With Concept EFBs,

- functions and
- Function blocks

can be defined:

Functions can be used anywhere where only a single output is required, especially booleans.

Function blocks can have several output parameters and can temporarily store data at runtime. See chapter *Keywords for Input and Output PINs, p. 58*.

Function Features

Functions are normally referred to in graphical languages (LD, FBD) as `xx.yy`, where `xx` is the section number and `yy` is the number of the function in this section.

Typical examples of such functions are comparisons **EQ_BOOL** or conversions **ABS_INT**, which return exactly one output parameter from one or more input parameters without needing to store any data temporarily.

Function Block Features

In the graphical languages (LD, FBD) it can be recognized as soon as as the properties dialog box corresponding to the block instance is visible, done with a double click on the block in Concept. The name of the instance of this block can be changed in the dialog box. The name automatically generated when creating the block is `FBI_xx_yy`, where `xx` is the number of the section and `yy` is a sequential number of the EFBs in this section.

A typical example of a function block is the timer **TON**, which has several output parameters and stores the current value at runtime over several PLC cycles.

In the non-graphical languages (ST, IL), the difference between functions and function blocks can be seen in the fact that a function block must be explicitly declared in the variable declaration (between `VAR` and `END_VAR`).

Implementation of the Interface

General

Before the actual programming of an EFB can be started, the interface between Concept and the C function in the EFB must be created.

Inputs/Outputs

The input and output PINs visible in Concept correspond to the C function parameters.

A maximum of 32 input and 32 output PINs are possible in an EFB.

Use data structures if you need more than 32 parameters in the EFB.

Extendable Inputs

Inputs may be defined as extendable if there can be more or less input PINs depending on your application. For special usage of extendable input parameters, please refer to the chapter *Usage of Extendable PINs, p. 57*.

Data Types

General

Only predefined data types can be used as parameters for an EFB.

These can be standard types such as **BOOL** or **INT**, as well as user defined types (as pointer to a data structure of this type).

The data type **ANY** and its specialized forms, such as **ANY_ELEM** or **ANY_BIT**, play a special role. They represent a data type which is defined when the EFB is instanced. See chapter *Data Type ANY and ANY_xxx, p. 53*.

IEC Data Types

The following table lists the data types according to IEC and their meaning in Concept.

IEC Data Type	Number of Bits	Numerical Range
BOOL	8	0, 1 or FALSE, TRUE
BYTE	8	Sequence of 8 bits (no numerical range)
WORD	16	Sequence of 16 bits (no numerical range)
INT	16	-32768...32767
DINT	32	-2147483648...2147483647
UINT	16	0...65535
UDINT	32	0...4294967295
REAL	32	$8.43*10^{-37}...3.36*10^{38}$
TIME	32	0... 4294967295 in [ms]

The IEC data types are available for use in C code.

Data Type ANY and ANY_xxx

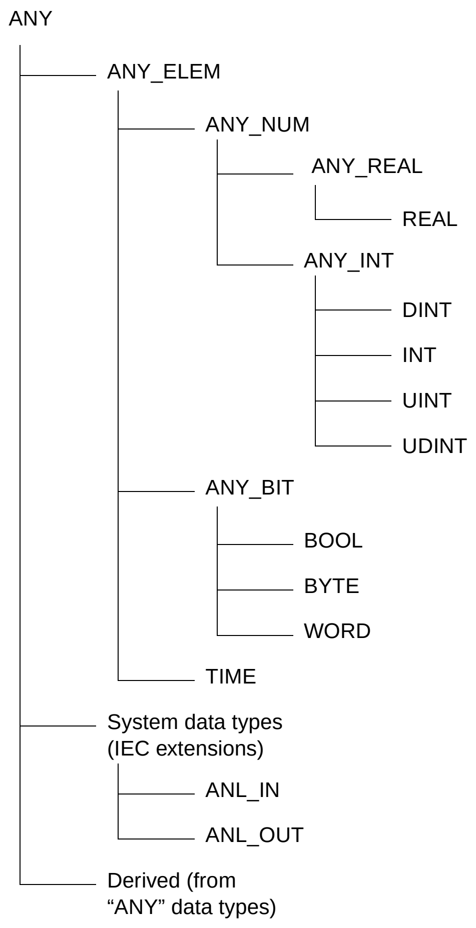
Differences between ANY and ANY_xxx

The generic data type **ANY** is assigned its data type when instanced in Concept. Only one data type can be fixed for one EFB. This means that extendible **ANY** PINs all have the same data types.

When using the data type **ANY_xxx**, multiple EFBs with the corresponding data types (e.g. ANY_INT includes DINT, INT, UINT and UDINT) are generated using one single source file (*.c).

Heirachy of the Data Types

The following graphic shows the heirachy of the data types from the general type **ANY**, its special forms **ANY_xxx** and the discrete IEC data types.



ANY

Data type **ANY** is a special form of a parameter for a block. If such a data type is used, the IEC data type used is decided when the block is instanced.

All PINs of the type **ANY** on the same EFB are assigned the same data type after instancing in Concept.

Additional Parameter `sizein`

When using the data type **ANY** the size of the data type is passed to the EFB as the additional parameter `sizein`.

Note: Compare the EFB **NYDT** from the SAMPLE library as an example.

Usage of the data type ANY_xxx

If one of the **ANY_xxx** data types are used as a parameter to a function block, multiple EFBs are created with multiple compiler passes in the `Makefile`. (For each IEC data type which can be derived from the given data type, a call to the compiler is generated).

The names of these derived EFB is derived from the EFB name and the C data type associated with it: **<EFB Name>_Data Type** e.g.**ADD_BOOL**

The various functions prototypes are listed in the prototype file (*.h).

A single source code (*.c) is used for the different derived EFBs.

The generic data type **ANY_xxx** can be used in the C function.

Note: Compare the EFB **GENDT** from the SAMPLE library as an example.

Compiler switch

The different compiler passes for the derived EFBs are differentiated with `<If defined> xxx_IMPLEMENTATION`

If special processing is necessary for the various data types of the PIN, the user can separate the code into different sections using conditional compilation.

e.g. `#ifdef xxx_IMPLEMENTATION (#else, #endif)`.

Note: Compare the EFB **SWGENDT** from the SAMPLE library as an example.

The following is a list of compiler switches which can be used depending on the IEC data types:

- `BOOL_IMPLEMENTATION`
- `BYTE_IMPLEMENTATION`
- `WORD_IMPLEMENTATION`
- `INT_IMPLEMENTATION`
- `UINT_IMPLEMENTATION`
- `DINT_IMPLEMENTATION`
- `UDINT_IMPLEMENTATION`
- `REAL_IMPLEMENTATION`
- `TIME_IMPLEMENTATION`

Which compiler switches are set for the generation of the block for different data types and which are evalutaed with `#ifdef, #else,#endif` can be seen in the `Makefile` for your library.

Usage of Extendable PINs

General

Passing data to an EFB using so called **extendable PINs** requires a special PIN definition with deviates from normal input PINs.

Only one extendable PIN can be used per EFB. This PIN must be graphically defined as the bottom input contact.

Procedure

The number of available inputs is visible in the special hidden parameter `nin`.

The macros in the following table are automatically generated in the prototype file. Use these macros to access the values passed to the block.

Macro	**Meaning**
`open_<pin>`	Starts the reading in of the inputs with names `<pin>`.
`next_<pin>`	Returns the value of the next input each time it is used.
`close_<pin>`	Finished the reading in of the inputs with names `<pin>`.

The term `<pin>` stands for the name of the parameter and is to be replaced by this when programming the EFB.

Please note that only the number of inputs as provided in `nin` are allowed to be read with `next_<pin>`, otherwise it causes a runtime error of the block.

Normally the reading is done in a FOR loop, using parameter `nin` as the exit condition.

Note: Compare the EFB **EXTINP** from the SAMPLE library as an example.

See also chapter *PIN Syntax, p. 47*.

Keywords for Input and Output PINs

Input

The keyword `Input` declares an input PIN.

Input PINs can be defined as extendable.

The syntax for the input or output PINs is explained in the Chapter *PIN Syntax, p. 47*.

Output

The keyword `Output` declares an output PIN.

The syntax for the input or output PINs is explained in the Chapter *PIN Syntax, p. 47*.

Internal state

Only function blocks can use the keyword `Internal State`. This keyword defines a data structure which is linked to the block. The structure is invisible to the user and is passed to the EFB with a pointer in the parameter list. Use this structure to define EFB internal data, storage for internal values, storage for old values etc.

The keywords `Rising Edge Detector`, `Falling Edge Detector` and `Internal State` add an entry to the EFB specific structure which can be accessed with a pointer.

Note: The variables for the `Internal State` structure behave like `unlocated variables` of the application and are explicitly initialized with `0`.

Rising/Falling Edge Detector

Both keywords `Rising Edge Detector` and `Falling Edge Detector` define edge detection. The corresponding declared parameters are resolved in the declaration of a boolean input variable and a variable of type `Internal State`, whose name is the name of the edge recognition with "_old" appended.

The result of this is that for the actual programming of the EFB, there is no difference if the edge recognition is defined in the definition file as

```
Rising Edge Detector     clk;
```

or

```
Input              BOOL   clk;
Internal State     BOOL   clk_old;
```

.

The keywords only serve to declare the required variables. The actual edge detection as well as the storing of the old status is the task of the the person who programs the EFB. This means that the detection of a rising edge requires al least the following code:

```
...
if (clk != clk_old && clk)
{
          ...
          clk_old = clk;
          return TRUE;
}
```

.

Var_InOut

This keyword allows user EFB to define PINs which can be simultaneously used as inputs and outputs in the EFB.

When this keyword is used, two PINs on the EFB are generated, one on the input side and a corresponding one on the output side, but in the prototype and in the source file, only one parameter is visible.

When the block is instantiated in a Concept editor, both sides are automatically linked to the same variable.

Note: You are not allowed to use the data type BOOL or ANY with this keyword.

Note: The position of `Var_InOut` pins can not be influenced. `Var_InOut` pins will always appear at the top position.

State output

This keyword serves to extend the output behaviour.

If an output is defined as type **BOOL** a 1x or 0x register may be connected to this PIN when the EFB is instanced in Concept.

The register values in the PLC may fluctuate during a single scan, meaning that the old value, as far as the EFB is concerned, will get lost.

This can be avoided by defining these outputs as **State Outputs**. A **State Output** frees Concept to store the attached value (even though it may be a 0x/1x register) as a variable so it doesn´t lose its value.

This is only necessary if the boolean value is not written to in every scan.

Note: Compare the EFB **STATEOUT** from the SAMPLE library as an example.

System Functions (API) Usage

General

The following system functions provided by Concept can be used in an EFB.

System Function	Call
AliGetProgStateEx	PTR_PROG_STATE CALL_CONV AliGetProgStateEx(void)
AliPutFbdError	void CALL_CONV AliPutFbdError(INT errno)
AliPutFbdError	void CALL_CONV AliPutFbdError(INT errno, INT param)

The prototypes for each function are in the prototype file `<efb>.h`.

Note: These system function may only be called in the main function of the EFB. It is not possible to call these functions in a subroutine as Concept cannot determin the EFB name and therefore the event logging does not work.

AliGetProg State()

AliGetProgState() returns a pointer to a data structure with system information (cold/ warm start, various clock signals, time of system start and a flag for a general system error).

Also see Chapter *Editing the prototype file, p. 42*.

AliPutFbdError()

AliPutFbdError() generates an online error message for the event log. Error messages which are sent to the event log are displayed with the given error number. The error numbers used by EFBs are predefined. They have special names which start with "EFB_USER_ERROR_1" up to "EFB_USER_ERROR_100" No user defined strings are allowed.

The definition for each error number is in the file `<efb>.err`. When this function is called:

- the name of the section where the block is called and
- the instance fo the EFB (only for graphical languages)

are automatically added to the error number.

If the function is called with the additional parameter **param**, this value is also displayed.

EFB Libraries

6

Overview

Introduction

This chapter provides an overview of handling function block libraries.

What's in this Chapter?

This chapter contains the following topics:

Creating EFB Libraries

Creating EFBs

First create the required function blocks as described in Chapter *Creating EFBs, p. 36*.

Also note Chapter *Code Limitations, p. 45*.

Make or Build

Function block libraries can be compiled using the submenu **Make** and also using the submenu **Build**.

When using the submenu **Make**, the dependencies of the files are checked before compiling the individual source files so that only those files are compiled that have been modified.

When using the submenu **Build** , all source files in a library are compiled regardless of the dependencies.

Note: Normally the compiler should find all dependencies and always compile what is necessary. It has become evident that not all dependencies are recognized and therefore files are ignored when compiling.

If you come across difficulties (unexpected error messages), it could be because not all necessary files were compiled. In this case, we recommend the lengthy, complete compilation.

Compile Library (Make)

Compile the library using the submenu **Library** → **Make**. The compilation of the library is also logged in a DOS window. After compilation, an editor window is shown `log.txt` with the logged messages. (When the option to automatically show the log file is set - otherwise you have to select it manually.)

Compile Library (Build)

Compile the library using the submenu **Library** → **Build**. The compilation of the library is also logged in a DOS window. After compilation, an editor window is shown `log.txt` with the logged messages. (When the option to automatically show the log file is set - otherwise you have to select it manually.)

Unlike **Make**, no dependencies for the individual files are considered, instead all files belonging to the library are compiled.

Installation of EFB Libraries

General

After a function block library has been compiled without syntax errors, it must be installed in Concept so that it can be used. This is done by invoking the **Install** function.

Install

Install the function block library after compiling with **Make** or **Build** (see Chapter *Creating EFB Libraries, p. 64*, using the submenu **Library** → **Install**.

A library can only be used in Concept after installation.

After the library is installed in Concept, a dialog box is shown where the user is asked if the library just installed should also be copied to an installation diskette. This function simplifies the installation of EFB libraries on several Concept stations and is described in Chapter *Installation of Libraries on Different Computers, p. 66*.

Note: The creation of an install diskette only works if the library and the setup files do not exeed the storage capacity of the diskette.

Installation of Libraries on Different Computers

General

The installation of libraries on different computers is carried out by creating an installation diskette on the system with Concept EFB installed.

Creating an installation diskette

After calling the sub-menu **Library** → **Install** (see Chapter *Installation of EFB Libraries, p. 65*), a dialog box is automatically shown which asks if an installation diskette should be created.

Answer this question with **Yes**.

Concept EFB then copies all files required for an installation to the diskette in the disk drive ´A:´.

The program `SETUPLIB.EXE` is also copied to the diskette in addition to the files belonging to the library.

Note: Make sure that enough free memory is available on the diskette.

The function is only suitable for creating installation diskettes. No other media are supported.

The function requires a correctly installed version of Concept EFB.

Installation

To install the library on another computer, start the program `SETUPLIB.EXE` on the target computer.

Note: The library can be installed on all computers with the same Concept version (e.g. 2.5). Which platform (s, m, xl) will be used is not important.

Testing Created EFB

General

The "trial and error" method is recommended for simple EFBs.

To do this, the error-free EFB library created as described in Chapter *Creating EFB Libraries, p. 64* is installed in Concept, the newly developed EFB is integrated in a test program and the program is tested using one of the two simulators delivered with Concept.

Checklist

As usual, a type of checklist should be prepared so the expected results can be compared with the actual results and all parameters variations checked.

Debugger

Complicated EFBs can also be tested in this way, but the error search can be difficult. Therefore it is recommended that the Debugger of the Borland C Compiler be used. A description of how this is done can be found in Chapter *Testing EFBs with the Debugger, p. 79*

Overview of the EFB Library Delivered

General

This chapter lists the main properties of the various function blocks in the SAMPLE EFB library delivered.

Properties of the function blocks from SAMPLE

The following table lists the function blocks from the SAMPLE library and their properties.

Name	Description	Type	Interface	Use of Istate	Other
BLOCK	Function Block with Rising Edge Detection	Function Block	Output	Edge Detector	Macro
DTY	Derived Data Type	Function Block	Input Output	-	Derived Data Type
EXTINP	Extensible input	Function Block	Input Output Extensible input Hidden	-	Macro
FUN	Function	Function	Input Output Hidden	-	-
FUNINEFB	Call of sub C-function within the EFB	Function	Input Output Hidden	-	-
GENDT	Generic Data Type	Function	Input Output Extensible input Hidden	-	Macro
GENSW	Derived generic data type (with use of compiler switch)	Function	Input Output Extensible input Hidden	-	Macro
INTSTATE	Internal state	Function Block	Input Output	Internal state	Macro

Name	**Description**	**Type**	**Interface**	**Use of Istate**	**Other**
NYDT	Generic Data Type ANY	Function	Input Output Hidden	-	Generic Data Type
ONLEVT	Online Event via AliPutFbdError		Input Output Hidden	-	PLC system function
STATEOUT	State Output	Function Block	-	State Output	Macro

Advanced Information for EFB Developers

7

Overview

Introduction

This chapter provides information for advanced EFB Developers.

What's in this Chapter?

This chapter contains the following topics:

Directory Structure

General

This chapter provides a representation of the directory structure created when developing a EFB library.

File and directory names, which have fixed names, are written in `CAPITAL LETTERS`.

Names, which are variable depending on the library names and EFB names, are written in the form `$<var>`.

The following variable names are used in the table "Directory structure":

Name	Meaning
$<DevPath>	Path of the Development directory
$<lib>	Name of the EFB library
$<efb>	EFB name

Note: The path name of the Development directory is limited to a maximum of 22 characters. Therefore, placing this directory in the root directory of a drive is strongly recommended. If an attempt is made to enter a path with more than 22 characters for the Development directory, an error message is generated.

Directory Structure

The following table shows the directory structure of a EFB library with a commentary on the respective directories and files.

Directory/File	Comment
$<DevPath>	
├── EFBLINK	Link Information for 16-bit and 32-bit EFBs
├── Inc	Include Directory
│ ├── EFB.ERR	error definitions
│ └── EFBH.H	type and value definitions+ system call functions
├── $<lib>	Library Directory
│ ├── BLD	Build Information for 16-bit and 32-bit EFBs
│ │ ├── DOS 16	
│ │ │ └── ASM	Assembler Code for 16-bit
│ │ └── WIN32	
│ │ └── ASM	Assembler Code for 32-bit
│ ├── $<efb>	Directory for an EFB
│ │ ├── $<efb>.FB	definition file
│ │ ├── $<efb>.C	C code source file of the EFB
│ │ ├── $<efb>.H	prototype file for the EFB
│ │ └── BACKUPxx.C	Backup file C code file (xx is the version number)
│ ├── $<lib>.DTY	data definition file for this library
│ ├── $<lib>.DTH	automatically generated C format file of $<lib>.DTY
│ ├── PROTO.H	automatically generated prototype include file
│ ├── MAKEFILE	Makefile for the compilation of the library
│ ├── LOG.TXT	error logging of Concept EFB
│ └── $<lib>.I	include for user information for all EFBs in this library
└── EFB.I	FP Marco and Constant definition (and further system specific includes)

Derived Data Types

General

In Concept EFB, it is possible to declare derived data types as in Concept.

Derived data types are a powerful tool for storing information in a program in a structured form.

A separate derived data type definition can be created for each EFB library (a group of at least one EFB).

Creating derived data types

If you declare your derived data types within Concept EFB, they are placed in a file with the name <lib>.DTY (see Chapter *Directory Structure, p. 72*). The data types defined in the global *.DTY file can also be used in Concept EFB. They are not integrated in the library.

After the installation of a library, the derived data types created in Concept EFB for this library are automatically made available in Concept i.e. they become global definded.

Editing <lib>.DTY

You can edit the file <lib>.DTY by selecting the menu item **Library** → **Derived Data Types** in Concept EFB (see Chapter *Library Menu, p. 29*). The file <lib>.DTY has the same structure as in Concept and corresponds to the structure of the programming language PASCAL.

Analysis of derived data types

If you open a DDT file (*.DTY) instead of a Concept project (*.prj) in Concept, you can edit and analyze the derived data types. The analyze function is useful for finding and solving problems related to derived data types.

Note: When declaring derived data types, the names of the data structures must be unique. If two different data structures exist with the same name but different structure in different libraries, errors will occur in Concept. Also see Chapter *Code Limitations, p. 45*.

This problem cannot be found by Concept EFB, because this global information is only available for all libraries after installing the library in Concept.

Use in Concept

If you wish to create an EFB that should access data structure elements which are already predefined in Concept (*.DTY files from the Concep/Lib directory), follow the following procedure.

Step	Action
1	Close Concept and Concept EFB
2	Search for the desired data type in the *.DTY files in the Concept\Lib directory
3	Using cut+paste copy the required data type from the global DTY to the libraries DTY. (Keep the name and structure).
4	Create the EFB.

File Extensions

Special File Extensions

In Concept EFB, special file extensions are used in addition to the normal file extensions used when programming in C.

File Extensions	Meaning
FB	Definition File. The definitions for an EFB are placed here according to the description in Chapter *Keywords for Input and Output PINs, p. 58*. This file defines the EFB´s interface with Concept.
TPL	TemPLate. This file describes the layout of an EFB as it will be displayed in Concept.
DTY	Derived Data TYpes. The definitions for derived data structures are placed in this file.
I	Include. This file contains general definitions for all function blocks belonging to a library.
DTN	This file is created using the DTY file and contains the names of all structures declared in the DTY file.
DTH	This file is the C format of the DTY file.

User Includes

User Includes

Using additional Includes, specific files (definitions, code, etc.) can be included in the EFB or in the library.

On the library level, it is the file `$<lib>`.I

> **Note:** As they are not Concept EFB standard files, User Include files are not processed by functions **Convert**, **Import** and **Move** in the **File** menu.

Floating Point Processor

Floating Point Processor

The floating point processor on the PLC (not on all PLCs) can be written to and read from using several macros in the file `EFB.I`.

The x87 processor (floating point processor) can be initialized with the macro `FP_INIT`.

At the end of the real operations, the status may be requested and checked with the following sequence:

```
INT status;
FP_GET_STAT (status)
if (status !=0) then
AliPutFbdError (FP_ERROR-status);
```

Deactivating and Reactivating Function Blocks

General

Deactivating or reactivating function blocks makes it possible for the developer to create and complete a library in steps.

Should the development on an EFB prove troublesome, the EFB can be excluded temporarily from the library without actually deleting the EFB.

Testing individual EFBs

In this way, all function blocks required for a library can be declared, but some are removed from runtime processing so they can be tested individually.

Note that any test project used to test the function blocks in Concept must be adjusted to the current activation or deactivation of the EFBs in the library.

Creating Context Sensitive Help (Online Help)

General

Concept provides context sensitive help for each EFB (Command Button **Help on Type** in Properties Dialog Box for the EFB in Concept). Concept does not have help textsfor user EFBs. However, you can create you own help texts for your EFBs.

File format

You can create your help files in the following formats:

- **.chm** (Microsoft Windows compiled HTML help file)
- **.doc** (Microsoft Word format)
- **.htm** (Hypertext Markup Language)
- **.hlp** (Microsoft Windows help file (16 or 32 bit format))
- **.pdf** (Adobe Portable Document Format)
- **.rtf** (Microsoft rich text format)
- **.txt** (plain ASCII text format)

Name

The name of the help file must correspond exactly to the name of the EFB (e.g. SKOE.ext).

The only exceptions are typed EFB names (e.g. SKOE_BOOL, SKOE_REAL etc.) In this case, the name of the help file is the EFB name without the type append (e.g. EFB Name: SKOE_BOOL has help file SKOE.ext).

Directory

The help file can be placed in the following directories:

- Concept help directory
- Concept library directory

Calling help

Concept follows the following procedure when calling help:

Phase	Description
1	Search for the classic help file **<Libname>.hlp** in the Concept help directory. **Result:** If the search is successful, the help file is shown, otherwise on to phase 2.
2	Search for the help file **EFBName.ext** in the subdirectory <Libname> in the Concept library directory. The help file is searched for in the following order: ● .hlp ● .chm ● .htm ● .rtf ● .doc ● .txt ● .pdf **Result:** If the search is successful, the help file is shown, otherwise on to phase 3.
3	Search for the help file **Libname.ext** in the subdirectory <Libname> in the Concept library directory. Order, see Phase 2. **Result:** If the search is successful, the help file is shown, otherwise on to phase 4.
4	Search for the help file **EFBName.ext** in the subdirectory <Libname> in the Concept help directory. Order, see Phase 2. **Result:** If the search is successful, the help file is shown, otherwise on to phase 5.
5	Search for the help file **Libname.ext** in the subdirectory <Libname> in the Concept help directory. Order, see Phase 2. **Result:** If the search is successful, the help file is shown.
6	The search ends after phase 5 or when the respective help file is found.

Testing EFBs with the Debugger

General

Testing EFBs in a library is also possible with the Turbo Debugger from Borland.

Refer to the respective documentation for information on how to operate the Debugger.

Calling the Debugger

To be able to test an EFB library with the Turbo Debugger, the EFB library must be generated with the option for debug information turned on and then installed in Concept. The Debugger can be invoked as follows:

```
C:\BC50\BIN\TD32.EXE C:\Concept\PLCsim32.EXE
```

when using the 32 bit simulator.

Note: After each change to one of the source files, the library must be compiled and installed again before a new Debugger session can be started.

Select EFB_FP32.DLL

In the debug window, use **F3** or menu item **View** → **Modules** to open a dialog box with a list of modules belonging to the simulator.

In the dialog box that is shown, always select the file `EFB_FP32.DLL` regardless of the name of the library created. Concept EFB uses this file name when installing an EFB library for the functions to be debugged.

This means, that only EFBs from one EFB library can be tested together in the Debugger!

Enter library directory

Select **Options** → **Source File Path** to enter the directory name for the EFB source file. After this entry is made, it is possible to debug the EFB at source code level in the Debugger, if you now enter a breakpoint and start the simulator (debugger command ´RUN´.

You can now start Concept and download your test application. The moment Concept invokes the test EFB during the first scan. The debugger will stop on your breakpoint.

Note: Concept will produce an error that communication with the PLC has been broken off.

Editor

8

Overview

Introduction

This chapter provides an overview of the menu commands for the integrated text editor in Concept EFB.

What's in this Chapter?

This chapter contains the following topics:

Introduction

General

This chapter describes the menu commands for the integrated text editor in Concept EFB, which is used to edit files.

The built-in editor contains all the required functions needed to edit Concept EFB files.

Menu bar

The editor has a menu bar for the various editing functions as described in the following chapters.

Status line

The lower border of the editor window contains a status bar, which shows the current cursor position as well as insert/overwrite mode. The cursor position shown makes it easy to find errors in soures when compiling libraries.

Syntax highlighting

Another editor function is the coloured highlighting of syntax elements for the programming language C. See the detailed information provided in Chapter *Syntax Highlighting, p. 88*.

File Menu

General

The menu **File** offers the following possibilities:

- saving the current document,
- saving the current document under another name and
- ending edit session.

Save

Using the submenu **Save**, you can save all changes to the current document since the last time the document was saved or since it was opened. The document remains open so you can continue working on it.

Save as...

Using the submenu **Save as**, you open the standard Windows dialog box to save the current document under another name. The original is not changed.

Exit

With the submenu **Exit**, you can exit the text editor. If you haven't saved the document since the last change was made, you will be asked if you want to save the changes, if you want to discard them or if you want to cancel exiting.

Edit Menu

General

The menu **Edit** offers the following possibilities:

- cutting from the current document,
- copying,
- inserting,
- deleting and
- moving to a certain line.

Cut

Using the submenu **Cut**, you delete the marked text from the document. The text is stored in the clipboard and can be inserted at another position.

Note: This menu command is only available if text is marked in the document

Copy

Using the submenu **Copy**, you copy the text marked in the document into the clipboard. The original is not changed.

Note: This menu command is only available if text is marked in the document.

Paste

Using the submenu **Paste**, you insert the contents of the clipboard at the current cursor position in the document or replace the text currently marked in the document.

Note: This menu command is only available if the clipboard contains text.

Goto line...

Using the submenu **Goto line...** you open a dialog box where you can enter a certain line in the current document that you want to jump to.

Search Menu

General

The menu **Search** offers the following possibilities:

- searching for text in the current document
- replacing it.

Find

Using the submenu **Find**, you open a dialog box used to find certain text. Various search criteria can be defined. A detailed description of the dialog box can be found in Chapter *Find Menu, p. 85*.

Find next

Using the submenu **Find next**, you can search using the criteria set in the dialog box again. The menu command is only active if the criteria for the search was already defined with the menu command **Search**.

Replace...

Using the submenu **Replace...** you open a dialog box where you can define the criteria to search and replace bits of text. A detailed description of this dialog box can be found in Chapter *Replace Menu, p. 86* .

Find Menu

General	You can get to the **Find** dialog box using the submenu **Search**. In this dialog box, you can enter the search text and the search parameters.
Find what	In the **Find what** text field, enter the search text or word. If text is marked in the document when calling the dialog box, the marked text is automatically entered in this field.
Match Whole Word Only	If this button is selected, the search text will only be found as whole words, but not as parts of longer words.
Match Case	If this check box is activated, only exact matches with the same capitalization as the text in the **Find What** text field are searched.
Mark All Matches	If this check box is activated, all texts found in the document are marked, regardless of the defined search direction.
Direction	There are two option buttons here which determine the search direction.
Up	If this option button is marked, the document is searched backwards from the current cursor position to the beginning for the search text.
Down	If this option button is marked, the document is searched from the current cursor position to the end.
Find next	This button activates the search for the search text entered or repeats a search that has already been carried out. If the search is successful, the text found in the document are marked, otherwise a message is given that the search text could not be found.
Cancel	This button ends the search and closes the dialog box.

Replace Menu

General	You can get to the **Replace** dialog box using the submenu **Search**. In this dialog box, you can enter the search text, the replace text and other search parameters.
Find what	In the **Find what** text field, enter the search text or word. If test is marked in the document when calling the dialog box, the marked text is automatically entered in this field.
Replace With	In this text field, enter the text that should replace the search text entered above.
Match Whole Word Only	If this button is selected, the search text will only be found as whole words, but not as parts of longer words.
Match Case	If this check box is activated, only exact matches with the same capitalization as the text in the **Find What** text field are searched.
Mark All Matches	If this check box is activated, all texts found in the document are marked, regardless of the defined search direction.
Direction	There are two option buttons here which determine the search direction.
Up	If this option button is marked, the document is searched backwards from the current cursor position to the beginning for the search text.
Down	If this option button is marked, the document is searched from the current cursor position to the end.
Find next	This button activates the search for the search text entered or repeats a search that has already been carried out. If the search is successful, the text found in the document are marked, otherwise a message is given that the search text could not be found.

Replace	With this command button, the search text marked in the document is replaced with the text you entered in the text field **Replace With**.
Replace All	This command button replaces all texts in the entire document that match the search text with the text entered in the text field **Replace With**.
Cancel	This button ends the search and closes the dialog box.

Syntax Highlighting

General

The integrated editor makes it easier for the user to program EFBs using colored highlighting of syntax elements for the programming language C.

File entries for color differences

The color differences of the individual elements are set according to entries in the file `TEXTEDIT.COL`, which is found in the Concept directory.

This file contains the assignments for colored highlighting of language elements for all document types used in Concept. The assignments are structured using keywords.

Structure of keywords

Keywords begin with a colon and are at the start of the line.

The keywords are followed by a color code, which determines the color used to display the language elements shown in the following line. The colored highlighting is valid until a new keyword line is found or until the file end.

Keywords

The following keywords are identified:

Keyword	Meaning
:StartExtension	The beginning of the definition for files with the following extension
:*	Comment within the file TEXTEDIT.COL
:Keywords	Terms to be shown in the following color
:Separators	Special characters to be shown in the following color Unlike keywords, individual characters are recognized here
:Group	Common groups to be shown in the following color As special feature, these groups can be extended if the first term is entered
:Comment	Comment definition in the document

Color code

Color codes can be the English names of the 16 standard colors for a text window. They are:

- Black, Gray,
- Blue, Lightblue
- Green, Lightgreen
- Red, Lightred
- Cyan, Lightcyan
- Magenta, Lightmagenta
- Yellow, Lightyellow
- Lightgrey, White.

Errors

9

Overview

Introduction

This chapter provides an overview of handling errors in Concept EFB.

What's in this Chapter?

This chapter contains the following topics:

Error messages, error correction

General

This chapter provides an overview of error messages given in Concept EFB and how to correct errors.

Finding compiler error messages

Error messages generated when compiling a function block library are written in the log file. This file can be viewed in an editor window using the menu item **Library** → **Log File** (also see Chapter *Library Menu, p. 29*).

The error messages noted in this file correspond to the errors generated by the Borland C compiler. For a detailed description of all possible messages, see the help provided by the Borland C compiler.

Compiler installation error

If you try to compile a library, but the installation path set using **Options** → **Options** does not refer to a valid compiler directory, the following error message is given:

```
Error: pro failed with -314
```

This means that the compiler needed to compile the library cannot be found.

Make and Build not possible

If both menu items **Make** and **Build** cannot be called, this normally means that only the DTY and FB files were defined but the files needed by the compiler are not present. Run **Generate files**.

Packer error

If you receive error messages from the Packer program, this is mostly because of either

- C functions not supported by Concept have been used in an EFB
- The DGROUP segment has been invoked in an EFB (see below).

Simulator crash

If the simulator crashes when testing EFBs (GPF), this is mostly because of a pointer error in the C code. The equivalent behavior to GPF on the PLC is the loss of communication followed by a Stop code error.

Recognizing Instructions with DGROUP Segment

General

The PLC operating system does not allow programs to use the DGROUP segment. In some cases, the Borland compiler uses this segment to store data.

The following instructions are affected:

- Operations with floating point numbers,
- Function calls for non-static functions and
- Larger switch() or case instructions.

In such cases, the instructions are compiled but the link procedure cannot be completed.

Finding Instructions

You can find these kinds of instructions in your program if you look at the assembler listing for the respective EFB.

Concept EFB creates a corresponding file for each EFB in the directory `BLD\DOS16\ASM` or `BLD\WIN32\ASM` after successfully carrying out a **Make** or **Build**.

Example

An example of instructions with a reference to the DGROUP segment could look like this:

```
1581 0A9C 9B D9 E1   fabs
```

```
1582 0A9F 9B D8 1E 000R FCOMP DWORD PTR DGROUP:S@
```

As you can see, the word DGROUP appears in line 1582.

Corrective Action

This problem can be corrected with the following changes:

- Calls for statically declared function are needed instead of the function calls,
- Floating point operations used in this segment, can be eliminated by changing the source file.

Example: The instruction

```
T = A + 10.0; // T and A are floating point values
```

in the DGROUP segment is used, because the intermediate result of `A+10.0` is stored in this segment.

This can be avoided if a variable is initialized with the value of the literal instead of adding a literal (e.g.: `float B = 10.0;`)

asm fnop;

The Borland C compiler can be forced to generate code which no longer uses the DGROUP segment by using the inline code assembler command `asm fnop;` (no operation for the x87) for the Floating Point Processor (x87).

Glossary

A

Active window

The window that is currently selected. Only one window can be active at any given time. If a window becomes active, the color of it’s title bar changes so it can be distinguished from other windows. Windows that are not selected are inactive.

Addresses

(Direct) addresses are memory areas on the PLC. They are found in the signal memory and can be assigned to input/output modules.

Direct addresses can be displayed/entered in the following formats:

- Standard Format (400001)
- Separator Format (4:00001)
- Compact Format (4:1)
- IEC Format (QW1)

ANL_IN

ANL_IN stands for data type "analog input" and is used to process analog values. The data type is assigned to the 3x references defined in the I/O connection list for the configured analog input module automatically and therefore can only be used with unlocated variables.

ANL_OUT

ANL_OUT stands for data type "analog output" and is used to process analog values. The data type is assigned to the 4x references defined in the I/O connection list for the configured analog output module automatically and therefore can only be used with unlocated variables.

ANY

In this version, "ANY" includes the elementary data types BOOL, BYTE, DINT, INT, REAL, UDINT, UINT, TIME and WORD and data types derived from them.

ANY_BIT

In this version, "ANY_BIT" includes data types BOOL, BYTE and WORD.

ANY_ELEM	In this version, "ANY_ELEM" includes data types BOOL, BYTE, DINT, INT, REAL, UDINT, UINT, TIME and WORD.
ANY_INT	In this version, "ANY_INT" includes data types DINT, INT, UDINT and UINT.
ANY_NUM	In this version, "ANY_NUM" includes data types DINT, INT, REAL, UDINT and UINT.
ANY_REAL	In this version, "ANY_REAL" includes data type REAL.
Application window	The window containing the workspace, the menu bar and the tool bar for the application program. The name of the application program is shown in the title bar. An application window can have several document windows.In Concept, the application window corresponds to a Project.
Argument	Same as current parameter.
Array variables	Variables assigned a defined derived data type with the help of the keyword ARRAY. An array is a collection of data elements of the same data type.
ASCII Mode	American Standard Code for Information Interchange The ASCII mode is used for communication with various host devices. ASCII works with 7 data bits.
Atrium	The PC based controller is on a standard AT board and can be operated in an ISA bus slot on a host computer. The module has a motherboard (requires an SA85 driver) with two slots for PC104 daughter boards. One PC104 daughter board is used as CPU and the other for INTERBUS control.

B

Backup File (Concept-EFB)	The backup file is a copie of the last source code file. The name of this backup file is "backup??.c" (it is assumed that you never have more than 100 copies of your source code file). The first backup file has the name"backup00.c". If you have made and saved changes to the definitions file, which don’t change the EFB , you can edit the source code file (**Object** → **Source**) and avoid the necessity of a backup file.

Base 16 Literal	In Concept Base 16 literals are used to enter whole number values in the hexadecimal system. The base must be labelled with the prefix 16#. The values cannot have a sign (+/-). Individual underlines (_) between numbers are not significant. Example 16#F_F or 16#FF (decimal 255) 16#E_0 or 16#E0 (decimal 224)
Base 2 Literal	In Concept Base 2 literals are used to enter whole number values in the binary system. The base must be labelled with the prefix 2#. The values cannot have a sign (+/-). Individual underlines (_) between numbers are not significant. Example 2#1111_1111 or 2#11111111 (decimal 255) 2#1110_0000 or 2#11100000 (decimal 224)
Base 8 Literal	In Concept Base 8 literals are used to enter whole number values in the octal system. The base must be labelled with the prefix 8#. The values cannot have a sign (+/-). Individual underlines (_) between numbers are not significant. Example 8#3_77 or 8#377 (decimal 255) 8#34_0 or 8#340 (decimal 224)
Binary connections	Connections between outputs and inputs on FFBs with data type BOOL.
Bit sequence	A data element consisting of one or more bits.
BOOL	BOOL stands for data type "boolean". The length of the data element is 1 bit (stored in 1 byte in memory). The value range for variables of this data type is 0 (FALSE) and 1 (TRUE).
Bridge	A bridge is an element used to connect networks. They allow communication between stations on two networks. Each network has its own token rotation order - the token is not passed through bridges.

BYTE	BYTE stands for data type "8 bit sequence". The entry takes place as Base 2 Literal, Base 8 Literal or Base 16 Literal. The length of the data element is 8 bits. A numerical value range cannot be assigned to this data type.

C

Call	Procedure used to start execution of an operation.
Clipboard	The clipboard is a temporary memory for cut or copied objects. These objects can be inserted in sections. Each time a new cut or copy is done, the previous contents of the clipboard are overwritten.
Coil	A coil is a LD element which transfers the state of the horizontal connection on its left side to the horizontal connection on its right side without any change. The state is stored in the corresponding variable/direct address during this process.
Compact Format (4:1)	The first number (the reference) is separated by a colon (:) from the following address, the leading zeros for the address are not given.
Constants	Constants are Unlocated Variables which are assigned a value which cannot be changed by the program logic (write protected).
Contact	A contact is an LD element that provides a state on the horizontal connection on the right. This state results from the boolean UND link of the state of the horizontal connection on the left and the state of the respective variable/direct address. A contact does not change the value of the respective variable/direct address.
Current parameters	Currently connected input/output parameters.

D

Data transfer settings

Settings to determine how information is transferred from your programming device to the PLC.

Data Types

The overview shows the hierarchy of the data types as used for inputs and outputs of Functions and Function Blocks. Generic data types are labelled by the prefix "ANY".

- ANY_ELEM
  - ANY_NUM
    ANY_REAL (REAL)
    ANY_INT (DINT, INT, UDINT, UINT)
  - ANY_BIT (BOOL, BYTE, WORD)
  - TIME
- System data types (IEC extensions)
- Derived (from 'ANY' data types)

DCP I/O Station

With a Distributed Control Processor (D908), you can set up a remote network with a master PLC above it. When using a D908 with a remote PLC, the master PLC handles the remote PC like a remote I/O station. The D908 and the remote PLC communicate via the system bus, which results in high performance with minimum effects on cycle time. Data exchange between the D908 and the master PLC takes place at 1.5 Megabit per second via the remote I/O bus. A master PLC can support up to 32 D908 processors.

DDE (Dynamic Data Exchange)

The DDE interface allows dynamic data exchange between two programs in Windows. The user can use the DDE interface in the advanced monitor to call custom visualization applications. With this interface, the user (i.e. DDE Client) can read data from the advanced monitor (DDE Server) and also write data to the PLC via the Server. The user can change data directly in the PLC while monitoring and analyzing the results. When using this interface, the user can create his own "Graphic-Tool", "Face Plate" or "Tuning Tool" and integrate it into the system. The tools can be written in any language e.g. Visual Basic, Visual C++ which support DDE. The tools are called if the user presses one of the buttons in the Advanced Monitor dialog box.Concept Graphic Tool: Using the DDE connection between Concept and Concept Graphic Tool, project signals can be shown in a clock diagram.

Declaration

Mechanism to determine a definition of a language element. A declaration normally includes the connection of a Label with a language element and the assignment of attributes such as data types and algorithms.

Definitions File (Concept EFB)	The definitions file contains general description information for the selected EFB and its formal parameters.
Derived data types	Derived data types are data types which are derived from the elementary data types and/or other derived data types. The definition of the derived data types is made in Concept’s Data Type Editor. A differentiation is made between global data types and local data types.
Derived Function Block (DFB)	A derived function block represents the call of a derived function block type. Details about the graphic form of the call can be found in the definition "Function block (instance)". Unlike calling EFB types, calls for DFB types are labelled with double vertical lines on the left and right side of the rectangular block symbol. The back of a derived function block type is designed in FBD language but only in the current version of the programming system. Other IEC languages cannot presently be used to define DFB types, also derived data types cannot be defined in the current version. A differentiation is made between local and global DFBs.
DINT	DINT stands for data type "double integer". The entry takes place as Integer Literal, Base 2 Literal Base 8 Literal or Base 16 Literal. The length of the data element is 32 bits. The value range for variables of this data type is -2 exp (31) to 2 exp (31) -1.
Direct Representation	A method of representing variables in the PLC program which can be used to directly derive the the assignment to the logical memory location - and indirectly to the physical memory location.
Document window	A window within an application window. More than one document window can be open in an application window at the same time. But only one document window can be active.Document windows in Concept are e.g. sections, the message window, the reference data editor and the PLC configuration.
DP (PROFIBUS)	DP = Decentral Peripheral
Dummy	An empty file consisting of a text header with general file information e.g. author, creation date EFB name, etc. The user has to complete this dummy file with additional entries.
DX Zoom	This property allows you to connect to a program object to monitor its data values and to change them if necessary.

E

Elementary Functions/ Function Blocks (EFB)	Name of Functions or Function Blocks, with type definitions which are not formulates in one of the IEC languages, i.e. their ends e.g. cannot be modified with the DFB editor (Concept DFB). EFB types are programmed in "C" and are prepared in precompiled form in libraries.
EN / ENO (Enable / Error display)	If the value of EN is equal to "0" when the FFB is called, the algorithms defined by the FFB are not executed and all outputs keep their previous value. In this case, the value of ENO is automatically set to "0". If the value of EN is equal to "1" when the FFB is called, the algorithms defined by the FFB are executed. After error free execution of these algorithms, the value of ENO is automatically set to "1". If an error occurs when executing these algorithms, ENO is automatically set to "0". The output behavior of the FFBs is independent of if the FFBs are called without EN/ENO or with EN=1. If the display if EN/ENO is turned on, the EN input must be used. Otherwise the FFB will never be executed. The configuration of EN and ENO is switched on or off in the function block properties dialog box. The dialog box is called using the menu command **Objects** → **Properties ...** or by double-clicking on the FFB.
Error	If an error is recognized when processing an FFBs or a step (e.g. invalid input values or timing error), an error message is given that you can view with the menu command **Online** → **event viewer...** For FFBs, the ENO output is set to "0".
Evaluation	The process with which a value for a function or for the outputs of a function block is calculated during the program execution.
Expression	Expressions consists of operator and operands.

F

FFB (Functions/ Function Blocks)	Collective term for EFB (Elementary functions/function blocks) and DFB (derived function blocks)
FIR Filter	(Finite Impulse Response Filter) Filter with finite impulse response
Formal parameter	Input/output parameters, used within the logic of an FFBs and which leave the FFB as inputs/outputs.

Function (FUNK)	A program organization unit that provides exactly one data element when executed. A function has no internal additional information. Multiple calls of the same function with the same input parameter values always return the same output values. Details about the graphic form of the function call can be found in the definition "Function block (instance)". Unlike calling function blocks, function calls have only one unnamed output because its name is the name of the function itself. In the FBD, each call is labelled by a unique Number using the graphic block; this number is created automatically and cannot be changed.
Function Block (instance) (FB)	A function block is a program organization unit, which calculates values for its outputs and internal variable(s) according to the functionality defined in its function block type description if it is called as a certain instance. All output values and internal variables for a certain function block instance remain from one function block call to the next. Multiple calls of the same function block instance with the same arguments (input parameter values) do not necessarily return the same output value(s). Each function block instance is graphically displayed using a rectangle block symbol. The name of the function block type is at the top middle within the rectangle. The name of the function block instance is also at the top, but outside of the rectangle. It is automatically generated when creating an instance, but can, if necessary, be changed by the user. Inputs are displayed on the left side, outputs on the right side. The names of the formal input/output parameter are displayed within the rectangle at the respective location. The description above for graphic display is principally valid for function calls and for DFB calls. Differences are described in the respective definitions.
Function block language (FBD)	One or more sections containing graphically displayed networks of functions, function blocks and links.
Function block type	A language element consists of: 1. the definition of a data structure divided into inputs, outputs and internal variables; 2. a set of operations executed with the elements of the data structure, if an instance of the function block type is called. This set of operations can be formulated either in one of the IEC languages (DFB Type) or in "C" (EFB Type). A function block type can can have multiple instances (calls).
Function counter	The function counter is used to uniquely label a function in a program or DFB. The function counter cannot be edited and is assigned automatically. The function counter always has the structure: .n.m n = Number of the section (consecutive number) m = Number of the FFB object in the section (consecutive number)

G

Generic data type	A data type which represents several other data types.
Generic literals	If the data type of a literal is not relevant, simply enter the value for the literal. In this case, Concept automatically assigns the literal a fitting data type.
Global derived data types	Global derived data types are available in each Concept project and are stored in the DFB directory directly under the Concept directory.
Global DFBs	Global DFBs are available in each Concept project and are stored in the DFB directory directly under the Concept directory.
Global Macros	Global Macros are available in each Concept project and are stored in the DFB directory directly under the Concept directory.
Group (EFBs)	Some EFB libraries (e.g. the IEC library) are divided in groups. This eases finding EFBs especially in extensive libraries.

I

I/O Connection List	In the I/O connection list, I/O modules and expert modules for the various CPUs are configured.
IEC 1131-3	International Standard: Programmable Logic Controllers - Part 3: Programming languages. March 1993.
IEC Format (QW1)	The first part of the address contains an IEC code followed by the 5-digit address: ● %0x12345 = %Q12345 ● %1x12345 = %I12345 ● %3x12345 = %IW12345 ● %4x12345 = %QW12345

IEC naming convention (code)	A code is a string of letters, numbers and underlines, which have to begin with a letter or underline (e.g. name of a function block type, an instance, a variable or a section). Letters from the national character set (e.g: ö,ü, é, õ) can be used, except in project and DFB names. Underlines are significant in codes; e.g. "A_BCD" and "AB_CD" are interpreted as different codes. Multiple underlines at the start or in a row are not allowed. Codes cannot have empty spaces.Capitalization is not significant; e.g. "ABCD" and "abcd" are interpreted as the same code. Codes cannot be keywords.
IIR Filter	(Infinite Impulse Response Filter) Filter with infinite impulse response
Initial step	The start step in a sequence. Each sequence must have an initial step defined. The sequence is started for the first call with the initial step.
Initial value	The value assigned to a variable when the program is started. The assignment of the value is made in the form of a literal.
Input bits (1x references)	The 1/0 state of input bits are controlled by the process data reaching from an input device to the CPU. **Note:** The x after the first number of the reference type represents a five digit memory location in application data memory, e.g. reference 100201 stands for an input bit at address 201 in signal memory.
Input parameter (input)	When calling an FFBs, provides the respective Argument.
Input word (3x references)	An input word contains information from an external source and are represented by a 16-bit value. A 3x register can also contain 16 consecutive input bits which are read into the register in binary or BCD (binary coded decimal) format.Note: The x after the first number of the reference type represents a five digit memory location in application data memory, e.g. reference 300201 stands for a 16 bit input word at address 201 in signal memory.

Instance Name	A label belonging to a certain function block instance. The instance name is used to clearly label a function block in a program organization unit. The instance name is created automatically, but can be edited. The instance name must be unique in the entire program organization unit, capitalization is not considered. If the entered name already exists, you will be warned and must select a new name. The instance name must correspond to then IEC naming conventions, otherwise an error message is given. Automatically created instance name always has the structure: FBI_n_m FBI = Function block instance n = Number of the section (consecutive number) m = Number of the FFB object in the section (consecutive number)
Instancing	Creating an instance.
Instruction (LL984)	When programming electric controllers, the user must implement operational coded instructions in the form of picture objects which are organized in recognizable contact form. The program objects designed, on the user level, are converted to OP codes that can be used by the computer during the loading process. The OP codes are decoded in the CPU and processed by the Firmware functions of the controller so that the desired control is implemented.
Instruction list (IL)	IL is a text language based on IEC 1131 in which operations such as conditional or unconditional function block and function calls, conditional or unconditional jumps, etc. are represented by instructions.
Instructions (IL)	Instructions are the "commands" used in programming language IL. Each instruction begins on a new line and is followed by an operator, if necessary with modifier and, if needed for the respective operation, by one or more operands. If several operands are used, they are separated by commas. A label can be placed before the instruction which is followed by a colon. The comment, if used, must be the last element in the line.
Instructions (ST)	Instructions are the "commands" used in programming language ST. Instructions must be concluded with a semicolon. Several instructions can be in a line (separated by semicolons).
INT	INT stands for data type "integer". The entry is made as Integer Literal, Base 2 Literal, Base 8 Literal or Base 16 Literal. The length of the data elements is 16 bit. The value range for variables of this data type ranges from -2 exp (15) to 2 exp (15) -1.

Integer Literals Integer literals are used to enter integer values in the binary system. The values can have a preceding sign (+/-). Single underlines (_) between the numbers are not significant.

Example

-12, 0, 123_456, +986

INTERBUS (PCP) To use the INTERBUS PCP channel and the INTERBUS process data preparation, the Concept configurator has a new I/O station type INTERBUS (PCP). This I/O station type is permanently assigned the INTERBUS connection module 180-CRP-660-01.

The 180-CRP-660-01 is different than the 180-CRP-660-00 only because of the clearly larger I/O area in the signal memory of the controller.

J

Jump Element of the SFC language. Jumps are used to jump over sections of the sequence.

K

Keywords Keywords are unique character combination, which are used as special syntactical elements such as those defined in Appendix B of the IEC 1131-3. All keywords in IEC 1131-3 and therefore those that can be used in Concept, are listed in Appendix C of IEC 1131-3. The listed keywords are not allowed to be used for anything else, e.g. not as variable names, section names, instance names, etc.

L

Ladder Diagram (LD)	Ladder diagram is a graphic programming language corresponding to IEC1131, which is oriented optically to the "current path" of a relay ladder diagram.
Ladder Logic 984 (LL)	In the terms Ladder Logic and Ladder Diagram, the word ladder (contact) refers to executions. Unlike a switching diagram, a ladder diagram is used by electronic technicians to draw a current circuit (using electrical symbols), which should show the process of events and not existing wires connecting the parts. A normal user environment to control actions of automation devices allows a ladder diagram interface, so that electronic technicians do not have to learn a programming language to implement a control program. The structure of the actual ladder diagram allows the connection of electric elements in a way which creates a control output depending on a logical current flow through the electronic objects used and represents the previously required condition of a physical electronic device. In simple form, the user environment is a video display processed by the PLC program application that sets up a vertical and horizontal grid where the program objects are assigned. The diagram receives current on the left side of the grid, and the current flows from left to right when connected with activated objects.
Landscape	Landscape means that the page, when looking at the printed text, is wider than high.
Language Element	Each basic element in one of the IEC programming languages, e.g. a Step in SFC, a Function block instance in FBD or the initial value of a variable.
Library	Collection of software objects which can be reused when programming new projects or even to create new libraries. Examples are the library of Elementaren Function Block Types. EFB libraries can be divided into Groups.
Link	A control or data flow connection between graphical objects (e.g. Steps in the SFC Editor, Function blocks in the FBD Editor) within a section, graphically represented as lines.

Literals	Literals are used to directly supply FFBs inputs, transition conditions, etc. with values. These values cannot be overwritten by the program logic (write protected). A differentiation is made between generic and typed literals. Additionally, literal are used to assign a constant a value or a variable an initial value. The entry is made as Base 2 Literal, Base 8 Literal, Base 16 Literal, Integer Literal, Real Literal or Real Literal with Exponent.
Local derived Data Types	Local derived data types are only available in a single Concept project and its local DFBs and are placed in the DFB directory under the project directory.
Local DFBs	Local DFBs are only available in a single Concept project and are placed in the DFB directory under the project directory.
Local Link	The local network link is the network that connects the local stations with other stations either directly or using a bus repeater.
Local Macros	Local Macros are only available in a single Concept project and are placed in the DFB directory under the project directory.
Local Network Station	The local station is the one that is currently being configured.
Located Variable	Located variables are assigned a signal memory address (reference addresses 0x, 1x, 3x,4x). The value of this variable is stored in signal memory and can be changed online with the reference data editor. These variables can be accessed with their symbolic names or with their reference address. All inputs and outputs on the PLC are linked with the signal memory. Access by the program of peripheral signals connected to the PLC only takes place using Located Variables. Access of the external side via Modbus or Modbus Plus interfaces on the PLC, e.g. by visualization systems are also possible using located variables.

M

Macro

Macros are created using the Software Concept DFB.

Macros serve to duplicate frequently used sections and networks (including the logic, variables and variable declarations).

A differentiation is made between local and global macros.

Macros have the following properties:

- Macros can only be programmed with the IEC programming languages FBD and LD.
- Macros contain only one section.
- Macros can have any complex section.
- From the point of view of the program, an instance of a macro, i.e. a macro inserted into a section, is no different to a section created conventionally.
- Calling DFBs in a macro
- Declaration of variables
- Usage of macro’s own data structures
- Automatic acceptance of variables declared in the macro
- Initialization values for variables
- Multiple instantiation of a macro in the whole program with different variables.
- The section name, the variable names and the data structure name can contain up to 10 different exchange markings (@0 to @9).

MMI

Man Machine Interface

Multielement-Variables

Variable, which are defined as STRUCT or ARRAY derived data types.

A differentiation is made between array variables and structure variables.

N

Network	A network is a connection of devices to a common data pathway, which communicate with each other with a common protocol.
Network node	A node is a device with an address (1...64) on a Modbus Plus network.
Node address	The node address serves as the unique reference for a network node in the routing path. The address is set directly on the node, e.g. with a rotary switch on the back of the module.

O

Operand	An Operand is a Literal, a Variable, a Function call or an Expression.
Operator	An operator is a symbol for an arithmetic or boolean operation to be executed.
Output parameter (output)	A parameter, with the result(s) of the evaluation of an FFB is returned.
Output/register bits (0x references)	An output/register bit can be used to control real output data through a control system output unit, or to define one or more discrete outputs in signal memory. Note: The x after the first number of the reference type represents a five digit memory location in application data memory, e.g. reference 000201 stands for an output or register bit at address 201 in signal memory.
Output/register bits (4 references)	An output register cna be used for storing numerical data (binary or decimal) Status RAM, or also for sending the data from the CPU to the output unit in the control system. Note: The x after the first number of the reference type represents a five digit memory location in application data memory, e.g. reference 400201 stands for a 16 bit output/register word at address 201 in signal memory.

P

Peer Processor	The peer processor logic processes token passes and the data flow between the Modbus Plus network and the PLC application logic.
PLC	Programmable Logic Controller
Portrait	Portrait format means that the page is higher than wide when viewing the printed text.
Program	The highest Program organization unit. A program is completely loaded onto individual PLCs.
Program cycle	A program cycle consists of reading the inputs, execution of the program logic and writing the outputs.
Program organizational unit	A Function, a Function block, or a Program. This term can either refer to a Type or an Instance.
Programming Device	Hardware and software that supports programming, configuration, testing, commissioning and the error analysis for PLC applications and decentralized system applications and also for source documentation and archiving. The programming device can also be also used for process visualization.
Project	General term for the highest level of a software tree structure which defines the top level project name of a PLC application. After definition of the project name, you can save your system configuration and control program with this name. All data that are generated during the creation of the configuration and the program belong to this project for this special automation task. General term for the complete set of programming and configuration information in the Project database, represented as source code which describes the automation of a system.
Project Database	The database in the Programming Device, which contains the configuration information for a Project.
Prototype File (Concept-EFB)	The prototype file contains all prototypes for the corresponding functions. As well, if available, a type definition of the internals.

R

REAL	REAL represents the data type "floating point number" It is input as aReal Literal or as Real Literal with Exponent. The length of the data elements is 32 bit. The value range for variables of this data type goes from 8.43E-37 up to 3.36E+38.
Real Literal	Real literals are for entering floating point numbers in a decimal system. Real literals are defined with the use of a decimal point. The values cam have a preceding sign (+/-). Single underlines (_) between the numbers are not significant. Example -12.0, 0.0, +0.456, 3.14159_26
Real Literal with Exponent	Real literals with exponent are for entering floating point numbers in a decimal system. Real literals with exponent are defined with the use of a decimal point. The exponent defines the power of 10 with which the preceding number is to be multiplied by to have its represented value. The values can have a preceding sign (+/-). Single underlines (_) between the numbers are not significant. Example -1.34E-12 or -1.34e-12 1.0E+6 or 1.0e+6 1.234E6 or 1.234e6
Redundancy system programming (Hot Standby)	A redundancy system consists of two identically configured PLC units which communicate with each other via redundancy processors. If a failure in the primary PLC occurs, the secondary PLC takes over control. Under normal conditions, the secondary PLC performs no control functions, it just checks the status information in order to recognize failures.

Reference	Every direct address is a reference, starts with a code which defines if it is an input or output and if it is a bit or a word. References which start with the code number 6, represent registers in extended memory of the status RAM, 0x Range = Coils 1x Range = Discrete Inputs 3x Range = Input Registers 4x Range = output registers 6x Range = Register in extended memory **Note:** The x after the first number of each reference type represents a five digit memory location in application data memory, e.g. reference 400201 stands for a 16 bit output/register word at address 201 in signal memory.
Register in extended memory (6x-Reference)	6x references are registers in the extended memory of the PLC. You can only use them in LL984 Application programs and only when using a CPU 213 04 or CPU 424 02.
Remote Network	Remmote programming in the Modbus Plus network allows maximum performance for data transfer and special requirements for links. Programming a remote network is simple. Additional ladder diagram logic does not have to be created to set up the network. Using respective entries in the Peer Cop Processor, all requirements for data transfer are handled.
RIO (Remote I/O)	Remote I/O defines the physical location of an I/O point control device with reference to the controlling processor. Remote inputs/outputs are connected to the control device with a communication cable.
RTU Mode	Remote Terminal Unit The RTU mode is used for the communication between the PLC and an IBM compatible Personal Computer. RTU works with 8 data bits.
Runtime error	Errors that occur during the processing the program on the PLC, for SFC objects (e.g. steps) or FFBs. They are e.g. value range overflows for counters or timing errors for steps.

S

SA85 Module	The SA85 module is a Modbus Plus adapter for and IBM-AT or compatible computer.
Section	A section can, for example, be used to describe the functions of a technological unit such as a motor. A Program or DFB has one or more sections. Sections can be programmed with the IEC programming languages FBD and SFC. Only one of the previously named programming languages can be used within one section. Each section has its own document window in Concept. For reasons of clarity, it makes sensible to divide a very large section into several smaller sections. The scroll bar is used to scroll within a section.
Separator Format (4:00001)	The first digit (the reference) is separated from the following five digit address with a colon (:).
Sequential Function Chart (SFC)	The SFC language elements allow the PLC program organization unit to be subdivided into a number of steps and transitions, which are connected with each other using directional connections. A number of actions belong to each step, and each transition is connected to a transition condition.
Serial Connections	The information is transferred bit-wise for serial interfaces (COM).
Source Code File (Concept-EFB)	The source code file is a normal C source file. After execution of the menu command **Library** → **files** create, this file contains an EFB-Code frame work in which you can enter code specific for the selected EFB. You must call the menu command **Object** → **Source**.
Standard Format (400001)	Directly after the first digit (the reference) is a five digit address.
Status bits	For each node with global input or specific input/output of peer cop data, there is a status bit. If a defined group of data has been successfully transferred within a configured time out, the corresponding status bit is set to 1. In other cases, the bit is set to 0 and all data in this group are cleared (set to 0).
Status RAM	The status RAM is the memory for all sizes, which is addressed via References (Direct Representation) in the user program. For example, discrete inputs, coils, input registers and output registers are in the Status RAM.

Step	SFC language element: A situation in which the behavior of a Program, with reference to its inputs and outputs, performs operations which are defined by the corresponding actions of a step.
Step Name	The step name is a unique reference of a step in a Program Organization unit. The step name is created automatically, but can be edited. The step name must be unique in the entire Program Organization Unit, otherwise an error message is given. The automatically created step name is as follows: S_n_m S = Step n = Number of the section (consecutive number) m = Number of the step in the section (consecutive number)
Structured Text (ST)	ST is a text language based on IEC 1131 in which operations such as function block and function calls, conditional execution of instructions, repeating instructions etc. are represented by instructions.
Structured Variables	Variables which are defined as being of a derived data type which has be defined with STRUCT (structure). A structure is a collection of data elements with generally different data types (elementary data types and/or derived data types).
SY/MAX	In Quantum control devices, Concept includes the provision for using SY/MAX I/O modules for RIO control with Quantum PLCs. The SY/MAX remote rack has a remote I/O adapter in slot 1 which communicates using a Modicon S908R I/O system. The SY/MAX I/O modules are for you to mark and include in the I/O list for the Concept configuration.
Symbol (Icon)	Graphical representation of different objects in Windows, e.g. drives, application programs and document windows.

T

Template File (Concept-EFB)	The template file is an ASCII file with layout information for the Concept FDB editor and the parameters for code creation.
TIME	TIME represents the data type "time". The entry is done as a Time Literal. The length of the data elements is 32 bit. The value range for variables of this data type goes from 0 to 2exp(32)-1. The unit for the data type TIME is 1 ms.
Time Literal	Allowable units for TIME are Days (D), Hours (H), Minutes (M), Seconds (S) and Milliseconds (MS) or combinations of these. The time must be marked with the prefix t#, T#, time# or TIME#. The "Overflow" of the highest value unit is allowed; e.g. the entry T#25H15M is allowed. Example t#14MS, T#14.7S, time#18M, TIME#19.9H, t#20.4D, T#25H15M, time#5D14H12M18S3.5MS
Token	The network "token" controls the temporary possession of the transmit rights for the individual nodes. The token cycles through the nodes in a sequential (increasing) address order. All nodes follow the token passing and can receive all data that is sent.
Traffic Cop	The traffic cop is a IO Map, which is generated from the user IO Map. The traffic cop is managed in the PLC and contains, in addition to the user IO Map, for example, status information for the I/O stations and modules.
Transition	The condition, by which the control changes from one or more preceding steps to one or more following steps according to a directed connection.

Typed Literal	If you want to defined the data type for a literal yourself, you can do this with the following construction: ’Data type name’#’Value of the Literals’. Example INT#15 (data type: Integer, Value: 15), BYTE#00001111 (Data Type: Byte, Value: 00001111) REAL#23.0 (Data Type: Real, Value: 23.0) For assignments to data type REAL, there are the following possibilities to define a value: 23.0. With a decimal point, the data type REAL is automatically assigned.

U

UDEFB	User defined basic functions/function blocks. Functions or Function blocks, which have been created in the programming language C and are available in Concept Libraries.
UDINT	UDINT represents the data type "unsigned double integer". The entry is made as Integer Literal, Base 2 Literal, Base 8 Literal or Base 16 Literal. The length of the data elements is 32 bit. The value range for variables of this data type goes from 0 to 2exp(32)-1.
UINT	UINT represents the data type "unsigned integer". The entry is made as Integer Literal, Base 2 Literal, Base 8 Literal or Base 16 Literal. The length of the data elements is 16 bit. The value range for variables of this data type goes from 0 to (2exp16)-1.
Unlocated Variable	Unlocated Variables are not assigned a status RAM address. They therefore do not use a Status RAM address. The values for these variables are stored internally by the system and can be changed with the reference data editor. These variables can only be accessed with their symbolic names. Signals which do not require access to peripherals, e.g. temporary results, system registers, etc. should preferably be declared as unlocated variables.

V

Variables	Variables are for data exchange within sections, between sections and between the Program and the PLC. Variables have alt least a variable name and a Data type. If a variable is assigned a direct address (reference), one refers to it as a Located Variable. If a variable is not assigned a direct address, one refers to it as an Unlocated Variable. If the variable is assigned a derived data type, one refers to it as a Multielement variable. Apart from this, there are alsoConstants and Literal.

W

Warning	If a critical condition is recognized during the execution of a FFBs or Step, (e.g. critical input value or time limit exceeded), a warning is generated. This can be viewed with the menu command **Online** → **Event Viewer...** For FFBs, the ENO output remains "1".
WORD	WORD represents the data type "Bit Sequence 16" The entry is done as Base 2 Literal, Base 8 Literal or Base 16 Literal. The length of the data elements is 16 bit. A numerical value range can not be assigned to this data type.

Index