|  |
| --- |
| **Model Report**  Version ● |
| |  | | --- | |  | | Date/Time Generated: | 29/05/2018 01:21:33 | | Author: | Luis Ruiz |   EA Repository : D:\SISTEMAS\AIMSERVICES\EA PROJECT\T3000.EAP |
| |  |  | | --- | --- | | **CREATED WITH** |  | |

**Table of Contents**

**1 T3000 CrossPlatform 4**

1.1 Business Process Model 4

1.1.1 Business Process Model diagram 4

1.1.2 <anonymous> 5

1.1.3 <anonymous> 5

1.1.4 <anonymous> 5

1.1.5 Note 5

1.1.6 $help://analysisdiagram.htm 6

1.1.7 $help://business\_process\_model\_pattern.htm 6

1.1.8 Business Objects 6

1.1.9 Business Workflows 6

1.1.10 Business Objects 6

1.1.10.1 Business Objects diagram 6

1.1.10.2 Note 7

1.1.10.3 Code Editor Detailed diagram 7

1.1.10.4 About FastColoredTextBoxNS 8

1.1.10.5 FastColoredTextBox 8

1.1.10.6 FastColoredTextBoxNS 9

1.1.10.7 Irony.Parsing 10

1.1.10.8 IronyFCBT 11

1.1.10.9 Parser 12

1.1.10.10 T3000.Grammar 13

1.1.10.11 .PRG or .PROG File 13

1.1.10.12 PRG Object 14

1.1.10.13 ProgramEditorForm 15

1.1.10.14 ProgramsForm 15

1.1.11 Business Workflows 17

1.1.11.1 Life Cycle of a Program diagram 17

1.1.11.2 .PRG or .PROG File 17

1.1.11.3 PRG Object 19

1.1.11.4 ProgramEditorForm 20

1.1.11.5 ProgramsForm 20

1.1.11.6 Edit Program 20

1.1.11.6.1 Edit Program diagram 28

1.1.11.6.2 PRG Object 30

1.1.11.6.3 Decode ProgramCode 30

1.1.11.6.4 Encode ProgramCode 32

1.1.11.6.5 Paint Code in Editor 35

1.1.11.6.6 Parse Internal Code 35

1.1.11.6.7 Parsing and Semantic Validations 35

1.1.11.6.8 Select user option 35

1.1.11.6.9 Set Code 35

1.1.11.6.10 Update Intenal Copy 36

1.1.11.6.11 Update PRG Object 36

1.1.11.6.12 Compile? 36

1.1.11.6.13 Exit? 36

1.1.11.6.14 Program has non-saved changes? 36

1.1.11.6.15 Send? 36

1.1.11.6.16 Code 36

1.1.11.6.17 Editable Copy on editTextBox.Text 37

1.1.11.6.18 Irony.FCBT Tokens 37

1.1.11.6.19 New ProgramCode 37

1.1.11.6.20 Begin 37

1.1.11.6.21 End 37

1.1.11.7 Process file into memory 37

1.1.11.8 Show ProgramPoints list 38

1.1.11.9 Click Edit ProgramCode Button 39

1.1.11.10 Load file .PRG or .PROG 39

1.1.11.11 Pregram's View - User Selection 39

# **T****3000 CrossPlatform**

*Package in package ''*

T3000 CrossPlatform

Version Phase 1.0 Proposed

Luis Ruiz created on 29/05/2018. Last modified 29/05/2018

## **B****usiness Process Model**

*Package in package 'T3000 CrossPlatform'*

Business Process Model

Version Phase 1.0 Proposed

created on 19/05/2018. Last modified 19/05/2018

### **Business Process Model diagram**

*Analysis diagram in package 'Business Process Model'*

Business Process Model

Version 1.0

Luis Ruiz created on 19/05/2018. Last modified 28/05/2018

1. Business Process Model

### **<anonymous>**

*Note in package 'Business Process Model'*

The Business Context package contains models of all involved stakeholders, mission statements, business goals and physical structure of the business "as-is".

<anonymous>

Version 1.0 Phase 1.0 Proposed

created on 19/05/2018. Last modified 19/05/2018

Extends

### **<anonymous>**

*Note in package 'Business Process Model'*

The Business Objects package contains a domain model of all objects of interest and their respective data.

<anonymous>

Version 1.0 Phase 1.0 Proposed

created on 19/05/2018. Last modified 19/05/2018

Extends

### **<anonymous>**

*Note in package 'Business Process Model'*

The Workflows package documents business processes, drawing on stakeholders, structures and objects defined in the Context and Object packages showing how these work together to provide fundamental business activities.

<anonymous>

Version 1.0 Phase 1.0 Proposed

created on 19/05/2018. Last modified 19/05/2018

Extends

### **Note**

*Note in package 'Business Process Model'*

The Business Process Model describes both the behavior and the information flows within an organization or system.

As a model of business activity, it captures the significant events, inputs, resources, processing and outputs associated with relevant business processes.

Note

Version 1.0 Phase 1.0 Proposed

created on 19/05/2018. Last modified 19/05/2018

Extends

### **$help://analysisdiagram.htm**

*Text in package 'Business Process Model'*

$help://analysisdiagram.htm

Version 1.0 Phase 1.0 Proposed

created on 19/05/2018. Last modified 19/05/2018

Alias View Further Examples

Extends

### **$help://business\_process\_model\_pattern.htm**

*Text in package 'Business Process Model'*

$help://business\_process\_model\_pattern.htm

Version 1.0 Phase 1.0 Proposed

created on 19/05/2018. Last modified 19/05/2018

Alias Read about Business Process Modeling

Extends

### **Business Objects**

*Package in package 'Business Process Model'*

Business Objects

Version 1.0 Phase 1.0 Proposed

created on 19/05/2018. Last modified 19/05/2018

### **Business Workflows**

*Package in package 'Business Process Model'*

Business Workflows

Version 1.0 Phase 1.0 Proposed

created on 19/05/2018. Last modified 19/05/2018

### **B****usiness Objects**

*Package in package 'Business Process Model'*

Business Objects

Version 1.0 Phase 1.0 Proposed

created on 19/05/2018. Last modified 19/05/2018

#### **Business Objects diagram**

*Analysis diagram in package 'Business Objects'*

Business Objects

Version 1.0

Luis Ruiz created on 19/05/2018. Last modified 22/05/2018

1. Business Objects

#### **Note**

*Note in package 'Business Objects'*

The Business Objects package contains representations of information, reports, data stores and other artifacts which are manipulated and used within the business workflows

Note

Version 1.0 Phase 1.0 Proposed

created on 19/05/2018. Last modified 19/05/2018

Extends

#### **Code Editor Detailed diagram**

*CompositeStructure diagram in package 'Business Objects'*

Explains how Code Editor (user control) is made, and how it works.

Code Editor Detailed

Version 1.0

Luis Ruiz created on 22/05/2018. Last modified 28/05/2018

1. Code Editor Detailed

#### **About FastColoredTextBoxNS**

*Note in package 'Business Objects'*

These are the main components of Code Editor, A.K.A FCBT or IronyFCBT.

Each component describes its parts, functionality and dependencies.

About FastColoredTextBoxNS

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 28/05/2018. Last modified 28/05/2018

Extends

#### **FastColoredTextBox**

*Class «UserControl» in package 'Business Objects'*

It's a fast colored (syntax highlighting) text box created by Pavel Torgashev.

Syntax highlighting:

Unlike RichTextBox, the component does not use RTF. The information about the color and type of symbols is kept only in the component. It means that the coloring of the component has to be redone every time when entering text. In this case, the event TextChanged is applied.

A Range object which contains the information about modified text range pass into the event TextChanged. It permits the highlighting of the altered text fragment only.

For the search of fragments of text which need to be colored, it is possible to employ overloaded method Range.SetStyle() which accepts search pattern (regular expression).

More information, and links to GitHub source code:

**WARNING: To all contributors / programmers. DO NOT update source code for FCBT from any external source, even if most recent.**

**This copy of FCBT source code, has many changes to make it crossplatform compliant, starting with the removal of every Windows API calls.**

FastColoredTextBox

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 22/05/2018. Last modified 28/05/2018

| **CONNECTORS** |
| --- |
| **Nesting** Source -> Destination  From: FastColoredTextBox : Class, Public  To: IronyFCBT : Class, Public |

#### **FastColoredTextBoxNS**

*Namespace «Namespace» in package 'Business Objects'*

Namespace for IronyFCBT control.

It contains a FastColoredTextBox with support for Irony grammar and parser.

FastColoredTextBox includes support for syntax highlighting and common code editor features as line numbering, styling, etc.

Irony Grammar and Parser, includes support for lexical analysis, syntax parsing and a lot more.

How it works...

Place a control IrontFCBT in a form. It will act like a specialized textbox!!

As you update its text property by writing or passing values, it triggers an event to parse de code, break it in tokens, while trying to catch regular expressions and choose which style should be assigned before actually painting the text in the control. That's how highlighting works.

Parsing and determining if the code is grammar compliant, has been added to this control, in order to make it automatic and parallel with syntax highlighting, thus helping the programmer to make real time decisions before completing a sentence.

More over, parsing process prepares the basis for semantic validations, by counting lines, marking jumps and other type of code that interrupts logical sequence.

FastColoredTextBoxNS

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 22/05/2018. Last modified 28/05/2018

| **INCOMING STRUCTURAL RELATIONSHIPS** |
| --- |
| Aggregation from «UserControl» IronyFCBT to «Namespace» FastColoredTextBoxNS  [ Direction is 'Source -> Destination'. ] |

#### **Irony.Parsing**

*Class in package 'Business Objects'*

Main class of Irony - .NET Compiler Construction Kit

Roman Ivantsov, 4 Jan 2008

Irony is a development kit for implementing languages on the .NET platform. It uses the flexibility and power of the C# language and .NET Framework 3.5 to implement a completely new and streamlined technology of compiler construction.

Unlike most existing yacc/lex-style solutions, Irony does not employ any scanner or parser code generation from grammar specifications written in a specialized meta-language. In Irony, the target language grammar is coded directly in C# using operator overloading to express grammar constructs. Irony's scanner and parser modules use the grammar encoded as a C# class to control the parsing process.

The Irony project is hosted on CodePlex.

Irony brings several principal innovations into the field of compiler construction. Like many parser-building tools in use today, Irony produces a working parser from grammar specifications. However, unlike the existing parser builders, Irony does not use a separate meta-language for encoding the target grammar. In Irony, the grammar is encoded directly in C# using BNF-like expressions over grammatical elements represented by .NET objects. Additionally, no code generation is employed - Irony's universal LALR parser uses the information derived from C#-encoded grammar to control the parsing process.

More info and wikis:

Understanding what is Irony .NET Compiler Construction Kit

Basics of Grammars, Parsing and AST.

A Brief Article and Sample for Irony Grammar Explorer (for debugging )

Spanish source: Creacion de Gramaticas en Irony

Elaborado por estudiantes del la Universidad de San Carlos de Guatemala de la carrera de Ingenieria en Ciencias y Sistemas

A course on Irony for dummies

Other

**NOTE: T3000 CROSSPLATFORM uses a nuget's package as the most recently updated form of Irony.**

Irony.Parsing

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 22/05/2018. Last modified 28/05/2018

| **CONNECTORS** |
| --- |
| **Dependency** «import» Source -> Destination  From: Irony.Parsing : Class, Public  To: IronyFCBT : Class, Public |

|  |
| --- |
| **Dependency** «instantiate» Source -> Destination  From: Parser : Class, Public  To: Irony.Parsing : Class, Public |

#### **IronyFCBT**

*Class «UserControl» in package 'Business Objects'*

Inherits FastColoredTextBox

IronyFCBT

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 22/05/2018. Last modified 28/05/2018

| **OUTGOING STRUCTURAL RELATIONSHIPS** |
| --- |
| Aggregation from «UserControl» IronyFCBT to «Namespace» FastColoredTextBoxNS  [ Direction is 'Source -> Destination'. ] |

| **INCOMING STRUCTURAL RELATIONSHIPS** |
| --- |
| Aggregation from Parser to «UserControl» IronyFCBT  [ Direction is 'Source -> Destination'. ] |

| **CONNECTORS** |
| --- |
| **Nesting** Source -> Destination  From: FastColoredTextBox : Class, Public  To: IronyFCBT : Class, Public |

|  |
| --- |
| **Dependency** «import» Source -> Destination  From: Irony.Parsing : Class, Public  To: IronyFCBT : Class, Public |

|  |
| --- |
| **Dependency** «import» Source -> Destination  From: T3000.Grammar : Class, Public  To: IronyFCBT : Class, Public |

| **ATTRIBUTES** |
| --- |
| Grammar : Irony.Parsing.Parser.Grammar Public  It holds the grammar and language for a Irony based package containing C# specification (based on eBNF) for Control Basic.  [ Is static False. Containment is Not Specified. ] |

|  |
| --- |
| Parser : Parser Protected  Alias: Internal Parser  [ Is static False. Containment is Not Specified. ] |

#### **Parser**

*Class in package 'Business Objects'*

Irony.Parsing.Parser protected instance.

Internal copy of parser, used for tokenization of Control Basic text code.

Parser object instances are initialized with a copy of Language specification. Irony provides both.

This is a common sample found in internet.

*LanguageData language = new LanguageData(grammar);*

*Parser parser = new Parser(language);*

Language is a state tree representation of all possible states allowed by a defined grammar. When you assign a language to the Irony Parser, you basically are allowing the parser to create a ParseTree and test any sequence of source code against that language (grammar specification).

*ParseTree parseTree = parser.Parse(sourceCode);*

*ParseTreeNode root = parseTree.Root;*

As in any testing, anything could we wrong. So parsetrees, enumerates all states well used, and stops when an syntax error is found.

That's the very basics of parsing.

T3000.Grammar, is our Control Basic translation of EBNF Rules for that language. Is packaged as a DLL for T3000. Source code included is based on C# but requires a lot of knowledge and practicing to become able to translate from EBNF, into BNF overload based operators and mechanisms for Irony Grammars.

Read more:

Irony .NET

Understanding BNF and EBNF representations for grammars

Parser

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 22/05/2018. Last modified 28/05/2018

| **OUTGOING STRUCTURAL RELATIONSHIPS** |
| --- |
| Aggregation from Parser to «UserControl» IronyFCBT  [ Direction is 'Source -> Destination'. ] |

| **CONNECTORS** |
| --- |
| **Dependency** «instantiate» Source -> Destination  From: Parser : Class, Public  To: Irony.Parsing : Class, Public |

| **ATTRIBUTES** |
| --- |
| Grammar : Irony.Parsing.Parser.Grammar Protected  Alias: IronyFCBT.Grammar  Allows a previously defined Grammar based on Irony to be specified as value  [ Is static False. Containment is Not Specified. ] |

#### **T3000.Grammar**

*Class in package 'Business Objects'*

T3000.Grammar

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 22/05/2018. Last modified 28/05/2018

| **CONNECTORS** |
| --- |
| **Dependency** «import» Source -> Destination  From: T3000.Grammar : Class, Public  To: IronyFCBT : Class, Public |

#### **.PRG or .PROG File**

*Object «Binary File» in package 'Business Objects'*

A structured byte sequenced file containing blocks of information for each type of control point.

A binary file of this type, has many sections, it size and content is briefly discussed here (only for the format version allowed by T3000 CrossPlatform right now)

**File Header: 3 bytes**

**Signature**: 1st and 2nd bytes, 0x55 and 0xFF respectively

**FileRevision**: 3rd byte. (Current File Revision is 0x06) This byte is used to set the FileVersion attribute in newly created instance of PRG (in-memory object).

*Warning: This definitions are only for Rev6 or Current file version format.*

public static bool IsNewVersion(byte[] bytes, int revision = CurrentFileRevision) =>

bytes.ToByte(0) == 0x55 &&

bytes.ToByte(1) == 0xff &&

bytes.ToByte(2) == revision; //version

**Possible values are defined here**

namespace PRGReaderLibrary

{

public enum FileVersion

{

Unsupported,

Dos,

Rev6,

Current = Rev6

}

}

Current format is the same as Rev6.

So there are only two formats defined, for T3000 Prg Binary File.

Signature for DOS versions are stored on bytes 27th to 30th, and compared with its string representation: "!@#$"

Signature for Rev6 or Current versions, are stored in 3rd byte, and its string representation is "Uÿ"

public const string DosSignature = "!@#$";

public const string Rev6Signature = "Uÿ"; //0x55 0xff

**File Structure and Lenght (Rev6)**

.PRG or .PROG File

Version 1.0 Phase 1.0 Proposed

created on 19/05/2018. Last modified 28/05/2018

#### **PRG Object**

*InputOutput «FC\_InputOutput» in package 'Business Objects'*

In-memory instance of class PRG containing lists for each type of control points and their values (control codes) if any.

You must be aware of PRG File Structure to understand this object.

See

Every section different from Header is represented as a Object List inside PRGReaderLibrary.PRG Class.

*#region Main data*

*public List<InputPoint> Inputs { get; set; } = new List<InputPoint>();*

*public List<OutputPoint> Outputs { get; set; } = new List<OutputPoint>();*

*public List<VariablePoint> Variables { get; set; } = new List<VariablePoint>();*

*public List<ProgramPoint> Programs { get; set; } = new List<ProgramPoint>();*

*public List<ControllerPoint> Controllers { get; set; } = new List<ControllerPoint>();*

*public List<ScreenPoint> Screens { get; set; } = new List<ScreenPoint>();*

*public List<GraphicPoint> Graphics { get; set; } = new List<GraphicPoint>();*

*public List<UserPoint> Users { get; set; } = new List<UserPoint>();*

*public List<TablePoint> Tables { get; set; } = new List<TablePoint>();*

*public Settings Settings { get; set; }*

*public List<SchedulePoint> Schedules { get; set; } = new List<SchedulePoint>();*

*public List<HolidayPoint> Holidays { get; set; } = new List<HolidayPoint>();*

*public List<MonitorPoint> Monitors { get; set; } = new List<MonitorPoint>();*

*public List<ScheduleCode> ScheduleCodes { get; set; } = new List<ScheduleCode>();*

*public List<HolidayCode> HolidayCodes { get; set; } = new List<HolidayCode>();*

*public List<ProgramCode> ProgramCodes { get; set; } = new List<ProgramCode>();*

*public CustomUnits CustomUnits { get; set; } = new CustomUnits();*

PRG Object

Version 1.0 Phase 1.0 Proposed

created on 19/05/2018. Last modified 28/05/2018

#### **ProgramEditorForm**

*Resource «resource» in package 'Business Objects'*

* Contains a instance of IronyFCBT user control, as container to edit code.
* Contains a copy of original plain text code (backup)
* Contains a local copy of plain text code for editing in FCBT control.
* Parses the textcode using Irony.Parsing and creates a List of Tokens from Parser and Grammar to show real time errors and warnings to programmer.
* Does automatic hightlightning of code based on FCBT control.
* **Compiles automatically after loading and editions, also before sending (save code into file). Compilation term used here means all these processes executed in sequence, in order to determine the code is ok, and can be translated:**

1. Lexical Analysis
2. Parsing (Syntax Analysis)
3. Semantic Analysis and Validations
4. Translation into ByteCodes (only on Save event).

ProgramEditorForm

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 28/05/2018

#### **ProgramsForm**

*Resource «resource» in package 'Business Objects'*

Contains a List of ProgramPoints and List of ProgramCodes.

Displays the list ProgramPoints in the gridview, attaching to each row a Command Button to trigger ProgramCode Edition.

ProgramsForm

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 28/05/2018

### **Business Workflows**

*Package in package 'Business Process Model'*

Business Workflows

Version 1.0 Phase 1.0 Proposed

created on 19/05/2018. Last modified 19/05/2018

#### **Life Cycle of a Program diagram**

*Analysis diagram in package 'Business Workflows'*

Details all processes, inputs and outputs in the life cycle of a Program (ProgramCode) inside T3000 Crossplatform.

Life Cycle of a Program

Version 1.0

Luis Ruiz created on 19/05/2018. Last modified 29/05/2018

1. Life Cycle of a Program

#### **.PRG or .PROG File**

*Object «Binary File» in package 'Business Objects'*

A structured byte sequenced file containing blocks of information for each type of control point.

A binary file of this type, has many sections, it size and content is briefly discussed here (only for the format version allowed by T3000 CrossPlatform right now)

**File Header: 3 bytes**

**Signature**: 1st and 2nd bytes, 0x55 and 0xFF respectively

**FileRevision**: 3rd byte. (Current File Revision is 0x06) This byte is used to set the FileVersion attribute in newly created instance of PRG (in-memory object).

*Warning: This definitions are only for Rev6 or Current file version format.*

public static bool IsNewVersion(byte[] bytes, int revision = CurrentFileRevision) =>

bytes.ToByte(0) == 0x55 &&

bytes.ToByte(1) == 0xff &&

bytes.ToByte(2) == revision; //version

**Possible values are defined here**

namespace PRGReaderLibrary

{

public enum FileVersion

{

Unsupported,

Dos,

Rev6,

Current = Rev6

}

}

Current format is the same as Rev6.

So there are only two formats defined, for T3000 Prg Binary File.

Signature for DOS versions are stored on bytes 27th to 30th, and compared with its string representation: "!@#$"

Signature for Rev6 or Current versions, are stored in 3rd byte, and its string representation is "Uÿ"

public const string DosSignature = "!@#$";

public const string Rev6Signature = "Uÿ"; //0x55 0xff

**File Structure and Lenght (Rev6)**

.PRG or .PROG File

Version 1.0 Phase 1.0 Proposed

created on 19/05/2018. Last modified 28/05/2018

#### **PRG Object**

*InputOutput «FC\_InputOutput» in package 'Business Objects'*

In-memory instance of class PRG containing lists for each type of control points and their values (control codes) if any.

You must be aware of PRG File Structure to understand this object.

See

Every section different from Header is represented as a Object List inside PRGReaderLibrary.PRG Class.

*#region Main data*

*public List<InputPoint> Inputs { get; set; } = new List<InputPoint>();*

*public List<OutputPoint> Outputs { get; set; } = new List<OutputPoint>();*

*public List<VariablePoint> Variables { get; set; } = new List<VariablePoint>();*

*public List<ProgramPoint> Programs { get; set; } = new List<ProgramPoint>();*

*public List<ControllerPoint> Controllers { get; set; } = new List<ControllerPoint>();*

*public List<ScreenPoint> Screens { get; set; } = new List<ScreenPoint>();*

*public List<GraphicPoint> Graphics { get; set; } = new List<GraphicPoint>();*

*public List<UserPoint> Users { get; set; } = new List<UserPoint>();*

*public List<TablePoint> Tables { get; set; } = new List<TablePoint>();*

*public Settings Settings { get; set; }*

*public List<SchedulePoint> Schedules { get; set; } = new List<SchedulePoint>();*

*public List<HolidayPoint> Holidays { get; set; } = new List<HolidayPoint>();*

*public List<MonitorPoint> Monitors { get; set; } = new List<MonitorPoint>();*

*public List<ScheduleCode> ScheduleCodes { get; set; } = new List<ScheduleCode>();*

*public List<HolidayCode> HolidayCodes { get; set; } = new List<HolidayCode>();*

*public List<ProgramCode> ProgramCodes { get; set; } = new List<ProgramCode>();*

*public CustomUnits CustomUnits { get; set; } = new CustomUnits();*

PRG Object

Version 1.0 Phase 1.0 Proposed

created on 19/05/2018. Last modified 28/05/2018

#### **ProgramEditorForm**

*Resource «resource» in package 'Business Objects'*

* Contains a instance of IronyFCBT user control, as container to edit code.
* Contains a copy of original plain text code (backup)
* Contains a local copy of plain text code for editing in FCBT control.
* Parses the textcode using Irony.Parsing and creates a List of Tokens from Parser and Grammar to show real time errors and warnings to programmer.
* Does automatic hightlightning of code based on FCBT control.
* **Compiles automatically after loading and editions, also before sending (save code into file). Compilation term used here means all these processes executed in sequence, in order to determine the code is ok, and can be translated:**

1. Lexical Analysis
2. Parsing (Syntax Analysis)
3. Semantic Analysis and Validations
4. Translation into ByteCodes (only on Save event).

ProgramEditorForm

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 28/05/2018

#### **ProgramsForm**

*Resource «resource» in package 'Business Objects'*

Contains a List of ProgramPoints and List of ProgramCodes.

Displays the list ProgramPoints in the gridview, attaching to each row a Command Button to trigger ProgramCode Edition.

ProgramsForm

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 28/05/2018

#### **Edit Program**

*Process «process» in package 'Business Workflows'*

Loads Editor with a copy of plain text Program Code.

Edit Program

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 28/05/2018

Alias ProgramsForm.EditCodeColumn

| **ELEMENTS OWNED BY Edit Program** |
| --- |
| Decode ProgramCode : Process «FC\_Process»  Decodes the byte sequence of ProgramCode into plain text "Control Basic"  PGRReaderLibrary.Utilities.Decoder class uses 3 helpers, preparing the semantic validations and as support to translating the code.   * EditorJumpInfo * EditorLineInfo * EditorTokenInfo   **Main method of this class is DecodeBytes()**. It returns in plain text a fully decoded Control Basic program. As input, receives a byte array that contains 2000 bytes for a single ProgramCode object.  Sample above is for initialization of internal copy of Program Text for a Editor FCBT Form.  *Decoder.SetControlPoints(Prg);*  *string ProgramText = Decoder.DecodeBytes(Codes[Index\_EditProgramCode].Code);*  *form.SetCode(ProgramText);*  **How it works**  All tokens references here are provided by reading byte to byte ProgramCode. This process happens before any lexical or syntax validations. Irony parsing is only used once the code is translated and put in editor.  DecodeBytes is a fully recursive and smart function, fully aware of when it has been called recursively, that returns a string (a part or whole of Control Basic program code), while doing so, it keeps record of a start offset and end offset of indexes inspected in the byte array while decoding.  For the first time when this function is invoked, it takes apart the first two bytes that represents real length of program bytescodes, although ProgramCode section is always 2000 bytes.  For the remaining bytes of ProgramCode, every byte is analyzed and compared to every predefined byte TOKEN values. For example: TOKEN value for REM is 0x1A. All tokens are predefined in enumerations according to their historical value in T3000 C++.  While some tokens are expected in a certain order, others need to be inspected in POSTFIX notation, especially EXPRESSIONS.  In order to achieve perfect translations from bytes to text, DecodeBytes uses special tokens as markers and delegates other translations to helper functions such as GetExpression, GetComment, GetElsePart... Do you see the pattern there? Right, every helper function starts with a "Get" prefix!  Markers are special tokens, delimitating what can be found ahead while translating the bytecodes into plain text. Finally everything gets concatenated in order. Some of them are:   * NUMBER (marker for a line number) * REM (marker for a comment statement) * IF THEN ELSE (and every variation of them) * ASSIGMENT (after an identifiers, EQ sign indicates that next step is to read an expression) * EOF   While other markers are very simple, just like a single byte command, or two bytes commands. Many markers indicates a special treatment of the content ahead. So Simple markers are translated immediately, and more complex ones call their helpers.  All identifiers, are checked against it proper type of ControlPoint arrays inside PRG object, in order to know if it exists and it is valid, and second but most important, to get its label or name.  Numeric literals also known as numeric constant values to differentiate from other numbers like LINE NUMBERING and others. Those receive specialized treatment. Four bytes are bit reversed into a double precision value, divided by 1000, and tested to see if the result is a whole number or it has decimals, just to get a proper string representation of the numeric value.  *constvalue.Text = GetConstValue(source, ref offset); //*  **GetExpression and ParseRPN2Infix are the second and third more important functions.**  GetExpression creates and parses the list of bytes of a expression into a list of tokens, expected to be in RPN (Reverse Polish Notation) also known as Postfix. Later on, ParseRPN2Infix, uses stacks and precedence of operators to get the real representation of the expression, in a infix notation. This is the very basic technique for evaluating of RPN expressions and the soul of all classes and method (helpers) that translate from bytes into human readable form of Control Basic code according to its syntax, A.K.A. Decoding. |

|  |
| --- |
| Encode ProgramCode : Process «FC\_Process»  Encodes a byte sequence of ProgramCode from plain text "Control Basic" contained in IronyFCBT user control.  PGRReaderLibrary.Utilities.Encoder class uses 3 helpers, preparing the semantic validations and as support to translating the code.   * EditorJumpInfo * EditorLineInfo * EditorTokenInfo   **Main method of this class is EncodeBytes()**.  It returns a static byte array of fully encoded Control Basic program. As input, receives a list of EditorTokens representing a previously parsed list of tokens with Irony.Parsing plus some modifications needed to add more info to each regular token from parsing.  Sample below is the real thing in ProgramsForm, where the encoding and updating of Prg object occurs, only when triggered the SEND event.  *private void Form\_Send(object sender, SendEventArgs e)*  e Object, receives a prepared copy of the list of EditorTokens directly from the editor.  The next C# code, encodes the byte sequence, calculates its size and update the ProgramCode and ProgramPoint respectively.  *//Init a copy of controlpoints*  *Encoder.SetControlPoints(Prg);*  *//ENCODE THE PROGRAM*  *byte[] ByteEncoded = Encoder.EncodeBytes(e.Tokens);*  *var PSize = BitConverter.ToInt16(ByteEncoded, 0);*  *Encoder.ConsolePrintBytes(ByteEncoded, "Encoded");*  *MessageBox.Show($"Resource compiled succceded{System.Environment.NewLine}Total size 2000 bytes{System.Environment.NewLine}Already used {PSize} bytes.", "T3000");*  *// MessageBox.Show(Encoding.UTF8.GetString(ByteEncoded), "Tokens");*  *Prg.ProgramCodes[Index\_EditProgramCode].Code = ByteEncoded;*  *//The need of this code, means that constructor must accept byte array and fill with nulls to needSize value*  *Prg.ProgramCodes[Index\_EditProgramCode].Count = 2000;*  *Prg.Programs[Index\_EditProgramCode].Length = PSize;*  **How it works**  All tokens references here are provided by a Preprocessed list of tokens cast into EditorTokens objects. A List generated not from the internal parser of editor, but from a specially separated parser inside ProgramEditorForm.  This process happens only when SEND menu option triggered. Meanwhile, the last version of program text is only temporary stored in Editor control.  **Encoder.EncodeBytes is very simple: just processing of a ordered**  **list of EditorTokens, BUT their helpers contained in ProgramEditorForm are not. These helpers could be the hardest methods to understand for the inexperienced programmer.**  **Real work is done by public void ProcessTokens()**  Every time there is a compilation, ProcessTokens recreates a list of preprocessed tokens ready to be encoded.  Assuming that you already read DECODE topic, (read it now please if you didn't ) I'm going to give you a quick overview of processes inside ProcessTokens and their helpers.  **ProcessTokens()** receives nothing as parameters but finally it reconstruct an internal list of EditToken. To do so, it walks a ParseTree object "\_parseTree" previously populated from Irony Parsing.  Tokens in \_parseTree are raw, as grammar and parser produce them. More over, some of this raw tokens are useless in the process like the starting terminal of parseTree "CONTROL\_BASIC" and temporary tokens.  While *walking* the parseTree using infix order, every simple token gets immediately added to the list with more information, contained in his properties:  *public class EditorTokenInfo*  *{*  */// <summary>*  */// Original text token from parsing*  */// </summary>*  *public string Text { get; set; }*  */// <summary>*  */// Associated Terminal Name from Grammar*  */// </summary>*  *public string TerminalName { get; set; }*  */// <summary>*  */// Token Type (1 Byte)*  */// Token size for Comment string*  */// </summary>*  *public short Type { get; set; }*  */// <summary>*  */// Token value (1 Byte)*  */// </summary>*  *public short Token { get; set; }*  */// <summary>*  */// Control Point index*  */// </summary>*  *public short Index { get; set; }*  */// <summary>*  */// Operators Precedence*  */// </summary>*  *public short Precedence { get; set; }*   * TerminalName: AS produced by Irony Parser. * Token: Byte corresponding value * Index: for ControlPoint searching * Text: Real name of TERMINAL constant to match enumeration values like REM, IFP, etc. * Type of Terminal * Precedence: for operators and functions.   Also after every token analysis, there is a counting (line numbers and their offsets), and stacking of branches (like jumps) to do all the semantic validations later on. ie: Suppose there is a GOTO followed by a number that indicates jump to line number 140, but there is no such a line number in the program.  Very well, creating the list would be easy if not for the RPN or postfix notation used for expressions, functions and some other type of sentences. RPN is needed to assure the same result either way of translation, especially for expressions where precedence of operators is vital. That´s why it is the preferred way of saving intermediate or fully compiled code for expressions.  **But things are not that easy.** The logic of ProcessTokens includes ***special appearance of a very obfuscated GetExpression helper*.** A method even more complicated than the opposite in DecodeBytes.  **Getting the job done. How to.**  **The power inside GetExpression Helper.**  The method GetExpression on Encoder class, takes a range of parser tokens contained in the internal parserTree of the class Encoder, to read it from a start offset and then try to create and return a list of EditTokens in RPN order, finishing its job when properly identifies the end of a expression, function or branch.  To do so, it requires two special collections:  *List<EditorTokenInfo> Expr = new List<EditorTokenInfo>();*  *Stack<EditorTokenInfo> Oper = new Stack<EditorTokenInfo>();*  Simplifying: A list of expressions and Stack of Operators, used to **implement a Shunting-Yard Algorithm** (thanks to Dijsktra).  Read more here:  The stack becomes a temporary storage for testing all stages of every single expression or function to become a full expression, reading from the parseTree.Tokens list, until a end marker is found: LF, THEN, EOF, REM or ELSE.  The list is used as final storage for all tokens read while walking the tree of EditorTokens in postfix order but according precedence of operators.  Some special modifications were allowed in the method to make it work with functions, considering also that function calls are part of more complex expressions.  After all that magic, a list of EditTokens (RPN version of expression) is returned to ProcessTokens and added to final list of preprocessed tokens.  Finally, remember those semantic validations?  Well, now the list of token is complete, EditorForm also has a complete list of Jumps, Lines Offsets and it's able to do those validations. When any semantic errors are found, every error is injected in the list of parseTree.ParserMessaages like any Syntax Error if any. ie.  *if (TokenType == PCODE\_CONST.UNDEFINED\_SYMBOL)*  *{*  *//There is a semantic error here*  *//Add error message to parser and cancel renumbering.*  *//Don't break it inmediately, to show all possible errors of this type*  *\_parseTree.ParserMessages.Add(new LogMessage(ErrorLevel.Error,*  *tok.Location,*  *$"Semantic Error: Undefined Identifier: {tok.Text}{System.Environment.NewLine}Check if PRG object is valid.",*  *new ParserState("Validating Tokens")));*  *ShowCompilerErrors();*  *Cancel = true;*  **This eases that when semantic errors occur, they are shown in the same control "Compilation Errors" together with syntax errors.**  In any case, successful or not, compilation ends with a list of EditorTokens, containing also semantic information, that EncodeBytes uses when triggered SAVE event, to get a final byte array for ProgramCode. In the last stage, EncodeBytes replaces line numbers with offsets of the byte array. |

|  |
| --- |
| Paint Code in Editor : Process «FC\_Process»  Painting on Editor + Syntax Higlighting  Shows the code on editor area with syntax highlighting. |
| Parse Internal Code : Process «FC\_Process»  Internal Parser of Irony FCBT receives a copy of Grammar and use it to decompose editTextBox TEXT into tokens.  It validates syntax errors and retrieves errors and warnings from parser. |
| Parsing and Semantic Validations : Process «FC\_Process»  Shows syntax or semantic errors when found.  Other way notifies 0 errors |
| Select user option : DefinedProcess «FC\_DefinedProcess» |
| Set Code : Process «FC\_Process»  Creates a Copy of Program Code in Irony.FCTB Texbox.Text property  Updates local copy Code. |
| Update Intenal Copy : Process «FC\_Process» |
| Update PRG Object : Process «FC\_Process» |
| Compile? : Gateway «Gateway» |
| Exit? : Gateway «Gateway» |
| Program has non-saved changes? : Gateway «Gateway» |
| Send? : Gateway «Gateway»  Update in-memory copy of ProgramCode |
| Code : InputOutput «FC\_InputOutput»  String Object container of original plain text form of code according to Control Basic Syntax (decoded). |
| Editable Copy on editTextBox.Text : InputOutput «FC\_InputOutput»  This property is also the input for internal parsing before paint code on container with syntax highlighting. |
| Irony.FCBT Tokens : InputOutput «FC\_InputOutput»  Parsed output from internal parser on Irony.FCBT |
| New ProgramCode : InputOutput «FC\_InputOutput»  Byte encoded instance of new program code. |
| Begin : Begin «FC\_Begin» |
| End : End «FC\_End» |

| **OUTGOING BEHAVIORAL RELATIONSHIPS** |
| --- |
| Object Flow from «process» Edit Program to «resource» ProgramEditorForm |

| **INCOMING BEHAVIORAL RELATIONSHIPS** |
| --- |
| ControlFlow from Click Edit ProgramCode Button to «process» Edit Program |

|  |
| --- |
| Name: Uses  Object Flow from «resource» ProgramsForm to «process» Edit Program |

##### **Edit Program diagram**

*Flow Chart diagram in package 'Business Workflows'*

Edit Program

Version 1.0

Luis Ruiz created on 19/05/2018. Last modified 29/05/2018

1. Edit Program

##### **PRG Object**

*InputOutput «FC\_InputOutput» in package 'Business Objects'*

In-memory instance of class PRG containing lists for each type of control points and their values (control codes) if any.

You must be aware of PRG File Structure to understand this object.

See

Every section different from Header is represented as a Object List inside PRGReaderLibrary.PRG Class.

*#region Main data*

*public List<InputPoint> Inputs { get; set; } = new List<InputPoint>();*

*public List<OutputPoint> Outputs { get; set; } = new List<OutputPoint>();*

*public List<VariablePoint> Variables { get; set; } = new List<VariablePoint>();*

*public List<ProgramPoint> Programs { get; set; } = new List<ProgramPoint>();*

*public List<ControllerPoint> Controllers { get; set; } = new List<ControllerPoint>();*

*public List<ScreenPoint> Screens { get; set; } = new List<ScreenPoint>();*

*public List<GraphicPoint> Graphics { get; set; } = new List<GraphicPoint>();*

*public List<UserPoint> Users { get; set; } = new List<UserPoint>();*

*public List<TablePoint> Tables { get; set; } = new List<TablePoint>();*

*public Settings Settings { get; set; }*

*public List<SchedulePoint> Schedules { get; set; } = new List<SchedulePoint>();*

*public List<HolidayPoint> Holidays { get; set; } = new List<HolidayPoint>();*

*public List<MonitorPoint> Monitors { get; set; } = new List<MonitorPoint>();*

*public List<ScheduleCode> ScheduleCodes { get; set; } = new List<ScheduleCode>();*

*public List<HolidayCode> HolidayCodes { get; set; } = new List<HolidayCode>();*

*public List<ProgramCode> ProgramCodes { get; set; } = new List<ProgramCode>();*

*public CustomUnits CustomUnits { get; set; } = new CustomUnits();*

PRG Object

Version 1.0 Phase 1.0 Proposed

created on 19/05/2018. Last modified 28/05/2018

##### **Decode ProgramCode**

*Process «FC\_Process» owned by 'Edit Program', in package 'Business Workflows'*

Decodes the byte sequence of ProgramCode into plain text "Control Basic"

PGRReaderLibrary.Utilities.Decoder class uses 3 helpers, preparing the semantic validations and as support to translating the code.

* EditorJumpInfo
* EditorLineInfo
* EditorTokenInfo

**Main method of this class is DecodeBytes()**. It returns in plain text a fully decoded Control Basic program. As input, receives a byte array that contains 2000 bytes for a single ProgramCode object.

Sample above is for initialization of internal copy of Program Text for a Editor FCBT Form.

*Decoder.SetControlPoints(Prg);*

*string ProgramText = Decoder.DecodeBytes(Codes[Index\_EditProgramCode].Code);*

*form.SetCode(ProgramText);*

**How it works**

All tokens references here are provided by reading byte to byte ProgramCode. This process happens before any lexical or syntax validations. Irony parsing is only used once the code is translated and put in editor.

DecodeBytes is a fully recursive and smart function, fully aware of when it has been called recursively, that returns a string (a part or whole of Control Basic program code), while doing so, it keeps record of a start offset and end offset of indexes inspected in the byte array while decoding.

For the first time when this function is invoked, it takes apart the first two bytes that represents real length of program bytescodes, although ProgramCode section is always 2000 bytes.

For the remaining bytes of ProgramCode, every byte is analyzed and compared to every predefined byte TOKEN values. For example: TOKEN value for REM is 0x1A. All tokens are predefined in enumerations according to their historical value in T3000 C++.

While some tokens are expected in a certain order, others need to be inspected in POSTFIX notation, especially EXPRESSIONS.

In order to achieve perfect translations from bytes to text, DecodeBytes uses special tokens as markers and delegates other translations to helper functions such as GetExpression, GetComment, GetElsePart... Do you see the pattern there? Right, every helper function starts with a "Get" prefix!

Markers are special tokens, delimitating what can be found ahead while translating the bytecodes into plain text. Finally everything gets concatenated in order. Some of them are:

* NUMBER (marker for a line number)
* REM (marker for a comment statement)
* IF THEN ELSE (and every variation of them)
* ASSIGMENT (after an identifiers, EQ sign indicates that next step is to read an expression)
* EOF

While other markers are very simple, just like a single byte command, or two bytes commands. Many markers indicates a special treatment of the content ahead. So Simple markers are translated immediately, and more complex ones call their helpers.

All identifiers, are checked against it proper type of ControlPoint arrays inside PRG object, in order to know if it exists and it is valid, and second but most important, to get its label or name.

Numeric literals also known as numeric constant values to differentiate from other numbers like LINE NUMBERING and others. Those receive specialized treatment. Four bytes are bit reversed into a double precision value, divided by 1000, and tested to see if the result is a whole number or it has decimals, just to get a proper string representation of the numeric value.

*constvalue.Text = GetConstValue(source, ref offset); //*

**GetExpression and ParseRPN2Infix are the second and third more important functions.**

GetExpression creates and parses the list of bytes of a expression into a list of tokens, expected to be in RPN (Reverse Polish Notation) also known as Postfix. Later on, ParseRPN2Infix, uses stacks and precedence of operators to get the real representation of the expression, in a infix notation. This is the very basic technique for evaluating of RPN expressions and the soul of all classes and method (helpers) that translate from bytes into human readable form of Control Basic code according to its syntax, A.K.A. Decoding.

Decode ProgramCode

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 28/05/2018

Alias PGRReaderLibrary.Utilities.Decoder.DecodeBytes

##### **Encode ProgramCode**

*Process «FC\_Process» owned by 'Edit Program', in package 'Business Workflows'*

Encodes a byte sequence of ProgramCode from plain text "Control Basic" contained in IronyFCBT user control.

PGRReaderLibrary.Utilities.Encoder class uses 3 helpers, preparing the semantic validations and as support to translating the code.

* EditorJumpInfo
* EditorLineInfo
* EditorTokenInfo

**Main method of this class is EncodeBytes()**.

It returns a static byte array of fully encoded Control Basic program. As input, receives a list of EditorTokens representing a previously parsed list of tokens with Irony.Parsing plus some modifications needed to add more info to each regular token from parsing.

Sample below is the real thing in ProgramsForm, where the encoding and updating of Prg object occurs, only when triggered the SEND event.

*private void Form\_Send(object sender, SendEventArgs e)*

e Object, receives a prepared copy of the list of EditorTokens directly from the editor.

The next C# code, encodes the byte sequence, calculates its size and update the ProgramCode and ProgramPoint respectively.

*//Init a copy of controlpoints*

*Encoder.SetControlPoints(Prg);*

*//ENCODE THE PROGRAM*

*byte[] ByteEncoded = Encoder.EncodeBytes(e.Tokens);*

*var PSize = BitConverter.ToInt16(ByteEncoded, 0);*

*Encoder.ConsolePrintBytes(ByteEncoded, "Encoded");*

*MessageBox.Show($"Resource compiled succceded{System.Environment.NewLine}Total size 2000 bytes{System.Environment.NewLine}Already used {PSize} bytes.", "T3000");*

*// MessageBox.Show(Encoding.UTF8.GetString(ByteEncoded), "Tokens");*

*Prg.ProgramCodes[Index\_EditProgramCode].Code = ByteEncoded;*

*//The need of this code, means that constructor must accept byte array and fill with nulls to needSize value*

*Prg.ProgramCodes[Index\_EditProgramCode].Count = 2000;*

*Prg.Programs[Index\_EditProgramCode].Length = PSize;*

**How it works**

All tokens references here are provided by a Preprocessed list of tokens cast into EditorTokens objects. A List generated not from the internal parser of editor, but from a specially separated parser inside ProgramEditorForm.

This process happens only when SEND menu option triggered. Meanwhile, the last version of program text is only temporary stored in Editor control.

**Encoder.EncodeBytes is very simple: just processing of a ordered**

**list of EditorTokens, BUT their helpers contained in ProgramEditorForm are not. These helpers could be the hardest methods to understand for the inexperienced programmer.**

**Real work is done by public void ProcessTokens()**

Every time there is a compilation, ProcessTokens recreates a list of preprocessed tokens ready to be encoded.

Assuming that you already read DECODE topic, (read it now please if you didn't ) I'm going to give you a quick overview of processes inside ProcessTokens and their helpers.

**ProcessTokens()** receives nothing as parameters but finally it reconstruct an internal list of EditToken. To do so, it walks a ParseTree object "\_parseTree" previously populated from Irony Parsing.

Tokens in \_parseTree are raw, as grammar and parser produce them. More over, some of this raw tokens are useless in the process like the starting terminal of parseTree "CONTROL\_BASIC" and temporary tokens.

While *walking* the parseTree using infix order, every simple token gets immediately added to the list with more information, contained in his properties:

*public class EditorTokenInfo*

*{*

*/// <summary>*

*/// Original text token from parsing*

*/// </summary>*

*public string Text { get; set; }*

*/// <summary>*

*/// Associated Terminal Name from Grammar*

*/// </summary>*

*public string TerminalName { get; set; }*

*/// <summary>*

*/// Token Type (1 Byte)*

*/// Token size for Comment string*

*/// </summary>*

*public short Type { get; set; }*

*/// <summary>*

*/// Token value (1 Byte)*

*/// </summary>*

*public short Token { get; set; }*

*/// <summary>*

*/// Control Point index*

*/// </summary>*

*public short Index { get; set; }*

*/// <summary>*

*/// Operators Precedence*

*/// </summary>*

*public short Precedence { get; set; }*

* TerminalName: AS produced by Irony Parser.
* Token: Byte corresponding value
* Index: for ControlPoint searching
* Text: Real name of TERMINAL constant to match enumeration values like REM, IFP, etc.
* Type of Terminal
* Precedence: for operators and functions.

Also after every token analysis, there is a counting (line numbers and their offsets), and stacking of branches (like jumps) to do all the semantic validations later on. ie: Suppose there is a GOTO followed by a number that indicates jump to line number 140, but there is no such a line number in the program.

Very well, creating the list would be easy if not for the RPN or postfix notation used for expressions, functions and some other type of sentences. RPN is needed to assure the same result either way of translation, especially for expressions where precedence of operators is vital. That´s why it is the preferred way of saving intermediate or fully compiled code for expressions.

**But things are not that easy.** The logic of ProcessTokens includes ***special appearance of a very obfuscated GetExpression helper*.** A method even more complicated than the opposite in DecodeBytes.

**Getting the job done. How to.**

**The power inside GetExpression Helper.**

The method GetExpression on Encoder class, takes a range of parser tokens contained in the internal parserTree of the class Encoder, to read it from a start offset and then try to create and return a list of EditTokens in RPN order, finishing its job when properly identifies the end of a expression, function or branch.

To do so, it requires two special collections:

*List<EditorTokenInfo> Expr = new List<EditorTokenInfo>();*

*Stack<EditorTokenInfo> Oper = new Stack<EditorTokenInfo>();*

Simplifying: A list of expressions and Stack of Operators, used to **implement a Shunting-Yard Algorithm** (thanks to Dijsktra).

Read more here:

The stack becomes a temporary storage for testing all stages of every single expression or function to become a full expression, reading from the parseTree.Tokens list, until a end marker is found: LF, THEN, EOF, REM or ELSE.

The list is used as final storage for all tokens read while walking the tree of EditorTokens in postfix order but according precedence of operators.

Some special modifications were allowed in the method to make it work with functions, considering also that function calls are part of more complex expressions.

After all that magic, a list of EditTokens (RPN version of expression) is returned to ProcessTokens and added to final list of preprocessed tokens.

Finally, remember those semantic validations?

Well, now the list of token is complete, EditorForm also has a complete list of Jumps, Lines Offsets and it's able to do those validations. When any semantic errors are found, every error is injected in the list of parseTree.ParserMessaages like any Syntax Error if any. ie.

*if (TokenType == PCODE\_CONST.UNDEFINED\_SYMBOL)*

*{*

*//There is a semantic error here*

*//Add error message to parser and cancel renumbering.*

*//Don't break it inmediately, to show all possible errors of this type*

*\_parseTree.ParserMessages.Add(new LogMessage(ErrorLevel.Error,*

*tok.Location,*

*$"Semantic Error: Undefined Identifier: {tok.Text}{System.Environment.NewLine}Check if PRG object is valid.",*

*new ParserState("Validating Tokens")));*

*ShowCompilerErrors();*

*Cancel = true;*

**This eases that when semantic errors occur, they are shown in the same control "Compilation Errors" together with syntax errors.**

In any case, successful or not, compilation ends with a list of EditorTokens, containing also semantic information, that EncodeBytes uses when triggered SAVE event, to get a final byte array for ProgramCode. In the last stage, EncodeBytes replaces line numbers with offsets of the byte array.

Encode ProgramCode

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 29/05/2018

Alias Encode Control Basic into ByteCodes

##### **Paint Code in Editor**

*Process «FC\_Process» owned by 'Edit Program', in package 'Business Workflows'*

Painting on Editor + Syntax Higlighting

Shows the code on editor area with syntax highlighting.

Paint Code in Editor

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 22/05/2018

##### **Parse Internal Code**

*Process «FC\_Process» owned by 'Edit Program', in package 'Business Workflows'*

Internal Parser of Irony FCBT receives a copy of Grammar and use it to decompose editTextBox TEXT into tokens.

It validates syntax errors and retrieves errors and warnings from parser.

Parse Internal Code

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 22/05/2018

##### **Parsing and Semantic Validations**

*Process «FC\_Process» owned by 'Edit Program', in package 'Business Workflows'*

Shows syntax or semantic errors when found.

Other way notifies 0 errors

Parsing and Semantic Validations

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 22/05/2018

##### **Select user option**

*DefinedProcess «FC\_DefinedProcess» owned by 'Edit Program', in package 'Business Workflows'*

Select user option

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 19/05/2018

##### **Set Code**

*Process «FC\_Process» owned by 'Edit Program', in package 'Business Workflows'*

Creates a Copy of Program Code in Irony.FCTB Texbox.Text property

Updates local copy Code.

Set Code

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 19/05/2018

Alias ProgramEditorForm.SetCode

##### **Update Intenal Copy**

*Process «FC\_Process» owned by 'Edit Program', in package 'Business Workflows'*

Update Intenal Copy

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 19/05/2018

##### **Update PRG Object**

*Process «FC\_Process» owned by 'Edit Program', in package 'Business Workflows'*

Update PRG Object

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 19/05/2018

| **OUTGOING BEHAVIORAL RELATIONSHIPS** |
| --- |
| Object Flow from «FC\_Process» Update PRG Object to «FC\_InputOutput» PRG Object |

##### **Compile?**

*Gateway «Gateway» owned by 'Edit Program', in package 'Business Workflows'*

##### **Exit?**

*Gateway «Gateway» owned by 'Edit Program', in package 'Business Workflows'*

##### **Program has non-saved changes?**

*Gateway «Gateway» owned by 'Edit Program', in package 'Business Workflows'*

##### **Send?**

*Gateway «Gateway» owned by 'Edit Program', in package 'Business Workflows'*

Update in-memory copy of ProgramCode

##### **Code**

*InputOutput «FC\_InputOutput» owned by 'Edit Program', in package 'Business Workflows'*

String Object container of original plain text form of code according to Control Basic Syntax (decoded).

Code

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 19/05/2018

##### **Editable Copy on editTextBox.Text**

*InputOutput «FC\_InputOutput» owned by 'Edit Program', in package 'Business Workflows'*

This property is also the input for internal parsing before paint code on container with syntax highlighting.

Editable Copy on editTextBox.Text

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 19/05/2018

##### **Irony.FCBT Tokens**

*InputOutput «FC\_InputOutput» owned by 'Edit Program', in package 'Business Workflows'*

Parsed output from internal parser on Irony.FCBT

Irony.FCBT Tokens

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 19/05/2018

##### **New ProgramCode**

*InputOutput «FC\_InputOutput» owned by 'Edit Program', in package 'Business Workflows'*

Byte encoded instance of new program code.

New ProgramCode

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 19/05/2018

##### **Begin**

*Begin «FC\_Begin» owned by 'Edit Program', in package 'Business Workflows'*

##### **End**

*End «FC\_End» owned by 'Edit Program', in package 'Business Workflows'*

#### **Process file into memory**

*Process «process» in package 'Business Workflows'*

Opens the file, tests and loads its parts (every control points and control codes into memory object instance of Prg)

If any errors, returns a "FileVersionNotImplementedException", finishind PRG Load.

This error, commonly means that the binary file is not a recognized format for PRG as defined in T3000 Crossplatform. Only format well documented and working is Current or Rev6.

Process file into memory

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 28/05/2018

Alias T3000Form.LoadPRG

| **OUTGOING BEHAVIORAL RELATIONSHIPS** |
| --- |
| Object Flow from «process» Process file into memory to «FC\_InputOutput» PRG Object |

| **INCOMING BEHAVIORAL RELATIONSHIPS** |
| --- |
| ControlFlow from Load file .PRG or .PROG to «process» Process file into memory |

|  |
| --- |
| Object Flow from «Binary File» .PRG or .PROG File to «process» Process file into memory |

#### **Show ProgramPoints list**

*Process «process» in package 'Business Workflows'*

Loads a list of Programs Control Points into a gridview, allowing user to interact with any row.

A ProgramPoint, is a control point, a subheader of information related to a ProgramCode. It contains: Label, Name, Lenght and Index.

Each index in a control point list acts as universal key to locate in counterpart section, the corresponding control code or value.

So, ProgramPoint[0] contains a subheader of information for the real code stored in ProgramCode[0].

Show ProgramPoints list

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 28/05/2018

Alias ProgramsForm.SetView

| **OUTGOING BEHAVIORAL RELATIONSHIPS** |
| --- |
| Object Flow from «process» Show ProgramPoints list to «resource» ProgramsForm |

| **INCOMING BEHAVIORAL RELATIONSHIPS** |
| --- |
| Name: Requires  Object Flow from «FC\_InputOutput» PRG Object to «process» Show ProgramPoints list |

|  |
| --- |
| ControlFlow from Pregram's View - User Selection to «process» Show ProgramPoints list |

#### **Click Edit ProgramCode Button**

*Event in package 'Business Workflows'*

User has clicked EDIT button of any row in the grid of program points.

Click Edit ProgramCode Button

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 28/05/2018

#### **Load file .PRG or .PROG**

*Event in package 'Business Workflows'*

User has selected menu option to load a PRG file into T3000 Crossplatform

Load file .PRG or .PROG

Version 1.0 Phase 1.0 Proposed

created on 19/05/2018. Last modified 28/05/2018

#### **Pregram's View - User Selection**

*Event in package 'Business Workflows'*

User has clicked or selected

Option Menu: Control - Programs (ALT + P)

Pregram's View - User Selection

Version 1.0 Phase 1.0 Proposed

Luis Ruiz created on 19/05/2018. Last modified 28/05/2018

Alias ProgramsForm