DynTFTCodeGen

-VCC-
v1.0 April-June 2019, v1.1 – v1.2 April 2020, v1.3 May 2020, v1.4 June 2020, v1.4.1 March 2021

Table of contents:

1. Introduction

DynTFTCodeGen is a tool, used for designing and generating initialization code and event handlers for DynTFT projects. It features a drawing board, an object inspector, a component palette and various dialog boxes for application and project level settings.

[Parts of the application, or other components can be named differently across this documentation, like "ObjectInspector", "Object Inspector", "Object inspector", "object inspector". It is similar for other descriptions like "Designtime", "Design time", "Design-time". They refer to the same thing. This is mostly for readability.]
The following introduction sections mention various parts of the application, which should be described later in other chapters of this documentation.

1.1 Features

- Component properties can be edited using an object inspector. Event handlers are also included.
- Components can be locked, so their position and size can't be accidentally changed. When a component is locked, it can't be dragged/moved or resized on the drawing board.
- Ability to preview the design using a color theme (.dyncol file) - see limitations below.
- Components are organized using screens, so they wouldn't be displayed all at once.
- Components belonging to a particular screen, can be set to be visually persisted across all screens (design-time only).
- Ability to drop on the drawing board, components which are usually part of other components, that do not work on their own: Items, RadioButton, TabButton, KeyButton, MessageBox.
- Component properties are organized in two categories: Runtime and Designtime. Runtime properties are found in component type definition, while Designtime ones are used for editing only.

1.2 Known issues

- Components are flickering when moved or resized on the drawing board.
- Can't properly move/resize a component while scrolling with the mouse wheel
- On particular OS setups, the application can display "Cannot focus a disabled or invisible

window." after starting.

- Sometimes, after closing the application, it can keep a CPU core to the maximum (not sure how to reproduce). This was encountered a few times during development, so it might be related to test builds only.

- When manually removing component registration from a component, in Project Settings window, it automatically removes its dependencies, no matter they were set as dependencies of another component or not. The generated code should not be affected.

1.3 Features, not bugs

Some parts of the application may behave a bit differently than expected. This is by design, either to optimize operations or to keep implementations simpler. Others might be considered incomplete implementations.

- Copying components using keyboard shortcuts, requires the drawing board to be focused (mouse over drawing board). This depends also on a setting to automatically focus the drawing board when being hovered with the mouse. If that setting is off, users will have to click on one or more components to focus the drawing board, prior to the copy operation.

- Ctrl-A selects only those components from current screen. Use Ctrl-Shift-A for all screens.

- Components are hidden while pasting or selecting (for a faster operation). Users may notice that components are hidden for a few milliseconds. Pasting requires new components to be created, so setting their position and size would require additional painting operations. Being hidden, some of these operations are skipped.

- Copy-paste operations do not include screen size. Only components are copied.

- When pasting components (all copied from the same screen), they are pasted in the current screen, regardless of their "ScreenIndex" property.

- When dragging multiple selected components, towards 0 (either left or top), their relative position changes as they are constrained to the visible area. Since there is no undo operation, this can be cancelled by right-clicking while the left mouse button is still down.

- Property values, which depend on compiler directives, are displayed in red in ObjectInspector

- When starting to drag a component, the component won't move for the first 5px (configurable) of mouse movement in any direction. This is to prevent accidental moving on MouseDown. Hold Alt key while dragging, to temporarily disable this feature.

1.4 Limitations

- There is no information about used memory of the DynTFT components, at design time. Please use a simulator, as provided in DynTFT examples.

- There is no live drawing for VirtualKeyboard (colors are set to default). It has fixed size. This is because of the internal complexity of this component. Since there is no pointer to each key button, the keyboard relies on the internal structure of DynTFT to keep track of all its buttons. DynTFTCodeGen does not use that structure, so it would require a different mechanism of keeping track of all these buttons.

- No Unicode support.

- No zoom support.

- No undo/redo support.

- Although Object Inspector and Component Palette can be resized, they can't be moved.

- Many things are hardcoded, so there is no support for user-defined components. Although parts of the application allow customization, it was not implemented for adding new components.

- Runtime Z order may not always match design-time Z order. Most of the time, components

should not be overlapped. More than that, there are some components (e.g. DynTFTComboBox, which bring some of their parts to front, on run time, changing design-time Z order).

- Components can only be brought to front or sent to back.
- Not all constants are available in ObjectInspector as choices for a particular property. Constants are displayed based on their data type.
- Some enum-like properties (e.g. ArrowDir, Direction, Orientation etc) are allowed to be set to all available constants of a component.
- Displayed components are not transparent, so if overlapping, they may look a bit different on simulator and hardware.
- Schema files are not validated, so expect crashes or bugs if manually edited.
- Components don't snap to a grid or to each other. Use the "Lock" property to avoid moving them accidentally.
- Property values are not displayed in bold in ObjectInspector, if different than defaults.
- Adding new tabs to a page control, or new radio buttons to a radio group can be done only by using their "Items" property. There is no special pop-up menu for these components.
- Unknown component properties are ignored when loading a project. They are discarded on save. The same for copy-paste. This may be encountered when having two or more instances of DynTFTCodeGen, installed in multiple different locations, each with its own set of different schema files.
- Changing direction/orientation of a ScrollBar, TrackBar or ProgressBar, does not automatically swap their Width and Height values.
- Switching from one component to another on the DrawingBoard does not automatically select the last focused property in ObjectInspector.
- The icon order in component Palette is hardcoded. Icons are also built-in.
- Colors from color themes, are not embedded in projects. They have to be manually loaded when needed. Color themes will have to be managed separately for each DynTFT project.
- When pasting components leads to duplicate names, only the object names are modified. Captions are kept.
- When pasting components from multiple screens, the current screen is not changed, but the paste operation succeeds.
- No built-in simulator. Users will have to manually rebuild their project simulators after every code generation.
- Double clicking on properties in ObjectInspector, is not implemented for all property types.
- Double clicking on events in ObjectInspector starts a 200ms timer to automatically create a new handler. The double click action has to be faster than 200ms, to create the handler.
- Double clicking on a component does not generate an event handler. Use the ObjectInspector.
- No support for vertical text orientation. This is a limitation of DynTFT library.
- The height of items (see "ItemHeight" property) of the Items component, has to be manually adjusted according to used font. The same for button height on PageControl.
- Empty RadioGroups and PageControls will generate uncompilable code. Always have at least one button. If manually creating buttons at run time is desired, either use a RadioGroup/PageControl with a button and remove it dynamically, or manually create the RadioGroup/PageControl at run time.
- Project file name is added to generated files, to warn users about overwriting with a different project. If there are two projects with the same name in different locations, no warning is given. Also, the project name is verified from DynTFTGUI.pas only.
- Handler names are verified as case sensitive. If renamed to a different case, they are added as new handlers. Old handler headers will have to be manually removed. The same for parameter list of these handlers, which have to match definitions from Schema files. If they don't match, new

ones are created.

- Unused/unassigned event handlers are removed from a DynTFTCodeGen project, only when closing the project. If they appear in generated code, they have to be manually removed.

- Handler implementation is added based on new added handlers with regard to existing handler headers in the generated file. If their implementation is manually removed, the new implementation is re-added, only if adding new handlers or removing their headers from the interface section of the DynTFTHandles.pas unit.

- Properties are organized into Runtime and Designtime categories in ObjectInspector. They are sorted by name in each category. That can't be changed.

- Some property names are hardcoded into application, like "ObjectName", "ScreenIndex", "Left", "Top", "Width", "Height", "CreatedAtStartup", "HasVariableInGUIObjects", "Locked", "ActiveFont" etc. Do not rename them in schema files, as the application will become unusable.

- Not all properties / data types are constrained in ObjectInspector. Be careful about what you input. Integer properties accept string values, to allow user-defined constants to be used. This is to allow assigning variables, constants, functions etc to a property, ar runtime, which DynTFTCodeGen should not know about. However, properties like "ObjectName", "ScreenIndex", "Left", "Top", "Width", "Height" are required by DynTFTCodeGen, and have to be assigned to constants at designtime.

- In ObjectInspector, properties of array type (e.g. "Strings", "Items^.String", "AllButtonsWidth" etc) display their values in a single line, without any blank between values. An items editor is available to edit such properties.

- Object names are not validated against existing event handler names or viceversa. Name collisions will be detected on compiling the DynTFT project.

- When the template files are changed (*.txt files from the OutputParts directory), for the handlers file, the changes have to be manually applied. This is because DynTFTCodeGen either generates this file from scratch or adds code to it, but does not keep it in sync with template files.

- When there are components for which no property is set and all of them are configured to have no variable in DynTFTGUIObjects file, a local variable is generated and left unused after assignment. This will cause a compiler notification about unused variables. The limitation comes from the Schema files, which only support one type of implementation, assuming that components will be customized and at least one property will have a value different than default.

- Although DynTFTCodeGen projects contain the list of available components, if loading a project in a DynTFTCodeGen instance, which has a different list of available components, the project may not be loaded properly.

- DynTFTCodeGen can work with up to 255 screens and will generate code for them. It does not take into account the number of screens the library supports. See *CDynTFTMaxComponentsContainer* constant from DynTFTTypes.pas unit.

- The colors used at design-time are 32-bit only. The code generator handles both 32-bit and 16-bit colors when generating output files. See Project Settings for options.

- DynTFTCodeGen does not control the color theme used by a project. The color theme has to be manually included in a DynTFT project, using the DynTFTColorTheme.inc file.

- There is no option to limit the number of recent files. If really needed, the DynTFTCodeGen.ini file can be edited using a text editor.

- DynTFTCodeGen does not manage DynTFT projects. Compiler directives, like *DynTFTFontSupport* have to be manually added to projects (Project Settings in Delphi/FreePascal and .pld files in mikroPascal). The same for extra units, e.g.: ExternalResources.pas, added in DynTFTGUIAdditionalUnits.inc.

- No full support for RTTI on all components. RadioGroups and PageControls require special code to add buttons, which cannot be binary encoded using the current format.

- RTTI for PIC24/dsPIC is not fully tested.

1.5 FAQ:

- Q: Does DynTFTCodeGen backup the projects it saves?

- A: No, it does not backup projects. It is recommended to use a version control software and keep track of as many changes as possible. It does create backups of the generated DynTFTHandlers.pas file, because it is expected that users edit this file using a source code editor.

- Q: When I copy the value of a property from one component, then paste it to another component, why does it set its value to "[SelectionInfo]"?

- A: The "[SelectionInfo]" string is part of the copy-paste format content when copying components. This indicates that a component was focused when pressing Ctrl-C, not the Object Inspector. Make sure not to move the mouse cursor outside Object Inspector when copying a property value. Also, see Application Settings for options about focusing the drawing board.

- Q: How do I add my custom component to the application?

- A: As mentioned in the "Limitations" section, there is no proper support for that. The application statically depends on DynTFT to draw components. You can add your schema file to the "\Schemas" directory and add an entry to DynTFTCodeGenInstalledComponents.ini, but there would be no drawing for the new component. Automatic design-time calculations won't be possible either (e.g. see the "ItemHeight" property of a RadioGroup).

- Q: Why the property for the name of a component is called "ObjectName" and not "Name"?

- A: "Name", if implemented, should be a runtime property and it wouldn't be practical to ensure it is unique. It may be useful mostly for debugging. See *BaseSchema.dynscm* file.

- Q: Why the property for font is "ActiveFont" and not simply "Font"?

- A: ActiveFont is a pointer to font information. It is unpractical to store the entire description of a font in a property, so a different name makes more sense.

- Q: Out of the four generated files, which one can be manually edited using an external editor?

- A: Only DynTFTHandlers.pas can be manually edited, so users can add code to the pregenerated handlers. The other three files (DynTFTGUI.pas, DynTFTGUIObjects.pas and DynTFTFonts.pas) are completely regenerated.

- Q: How to swap two screens or move a screen to a different index?

- A: Hover the tab buttons of the list of screens with the mouse, hold the Ctrl key and start scrolling with the mouse wheel.

- Q: Can the settings of the default font be changed?

- A: Yes, just add it to the list in the Project Setting dialog, then change the settings. This is useful when using a custom TFT library.

- Q: If the generated code is smaller and less RAM is used when the "HasVariabileInGUIObjects" property is set to false for most components, why isn't this the default value?

- A: It is easier for users to find a component by its object name. For components which generate an event and have an assigned event handler, the Sender parameter of the handler can be used to get a pointer to that component. They can have the "HasVariabileInGUIObjects" property set to False. It requires typecasting though.

- Q: Where should I look first, if something behaves different than expected?
- A: There are general application settings and project-level settings, found under the Tools menu. In addition to that, there are screen options under the Screens pop-up menu. There are also various design-time component properties, which configure component behavior. Hints/tooltips will pop up all over the application. Unfortunately, not all features are configurable.

- Q: What should I do if I find a bug?
- A: Before reporting a bug, please see if it is already mentioned in one of the "Known issues", "Features, not bugs" or "Limitations" sections above. Do not send me or publicly post projects (*.dyntftcg files), as they might contain information you don't want to give away. If a minimal project is required to report a bug, please open it using a text editor, and look for stuff you might want to remove. When reporting a bug, it is desired to mention the steps used to reproduce the bug, by having the first step as "open the application", followed by "click here", "click there"-like steps. Also, if a project is already corrupt (either by DynTFTCodeGen or manually edited using a text editor), it has no value in reproducing a bug, because the application is not designed to handle corruptions. However, it is very useful to have the steps about how a good project can be corrupted by the application. If you are editing schema files, bug reports may have little value, because these are not validated, so the application may not work properly.

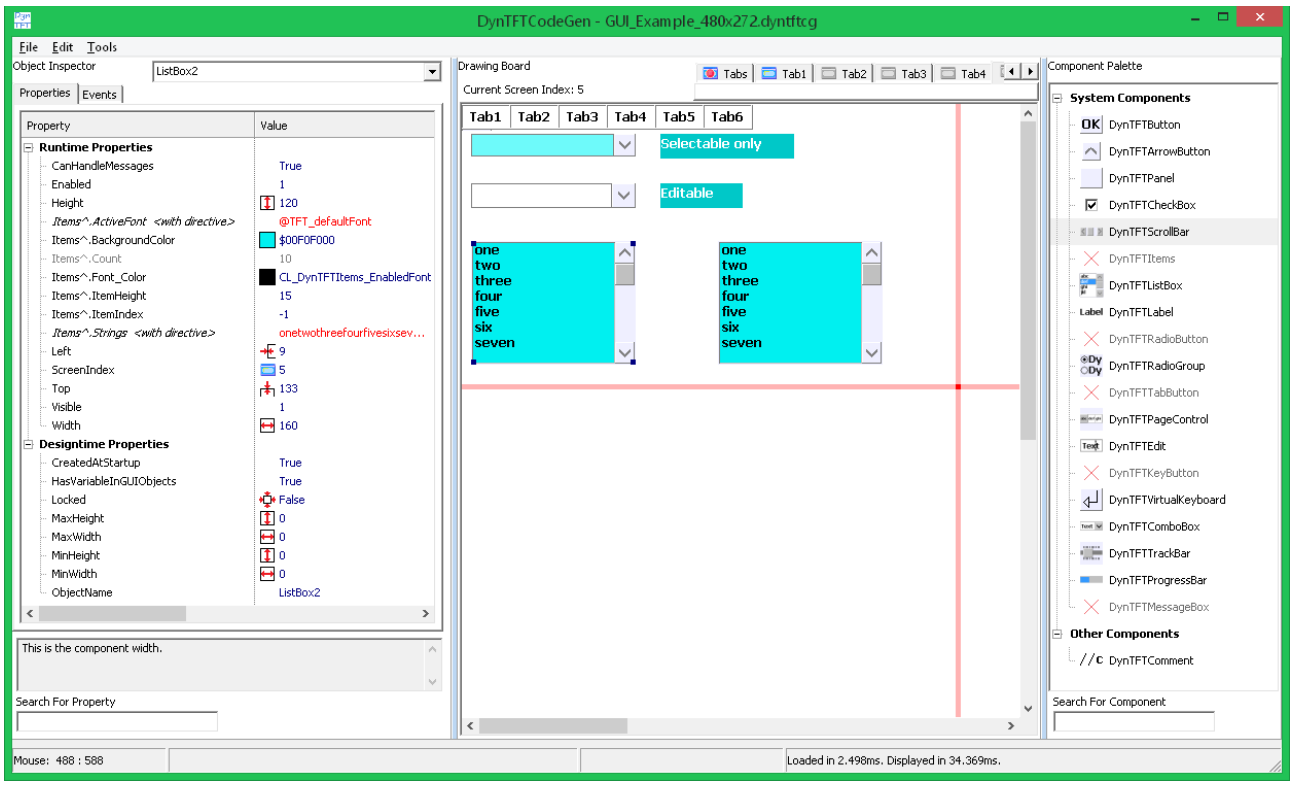
- Q: Can I use an already existing DynTFTHandlers.pas file in a project in which DynTFTCodeGen generates this file?
- A: Yes, if the code generation symbols are manually added on functions/procedures (also notice the difference //CodegenSym:header vs. //CodegenSym:handler) and the headers match the string format that comes from schema files. Probably, the easiest way is to let DynTFTCodeGen generate an empty file, then you can add your code to it.

- Q: Can I use "OnClickUser" event on components?
- A: At the moment of writing this, DynTFT library does not implement this event. Please use "OnMouseUpUser" event.

1.6 Application overview

DynTFTCodeGen is a simple application, with a main window and a few configuration windows. From the component palette, with currently implemented DynTFT components, users add components to the drawing board, by dragging them. They can be moved or resized on the drawing board, either by mouse or by keyboard. Various properties and event handlers can be set from the object inspector.

Components are organized into screens, so they won't be displayed all at once. Screens can be active or inactive, and persisted or not persisted. Active screens match the "Active" field of the "TDynTFTScreenComponentInfo" field, which controls which components to be painted, based on their "ScreenIndex" property/field. There are cases when users want components from a particular screen to be visible across all screens. For this, screens can be set to be persisted, so their components will be visible regardless of the current selected screen. For example, in the following screenshot, the first screen, called "Tabs" is set to persisted (notice the red pin icon over the (blue) screen icon), so the PageControl (the top most visible component on the drawing board) is visible, although the current screen is set to "Tab5". This is the intended behavior at run time, when this PageControl belongs to an active screen (at index 0), and is used to activate / deactivate the other screens. When changing current screen, components are hidden and displayed, based on their "ScreenIndex" property.



The Object Inspector is a key–value list (or property name – property value list), with all available properties of a component, defined by the schema files. It also features a property description box. Based on their data type, properties can be modified from the Object Inspector by various local editing components (Edit, ComboBox, ColorBox) or an items editor in case of array-like properties.

The main menu is kept simple, featuring File, Edit and Tools. There are also pop-up menus for drawing board, screens and components. The File menu is used for opening and saving projects, the Edit menu is organized into component and screen operations, while the Tools menu contains operations like application settings, project settings, loading color theme and generating code.

2. Included files

The application comes as a single executable, *DynTFTCodeGen.exe*, with various configuration files. It uses two ini files: *DynTFTCodeGenInstalledComponents.ini* and *DynTFTCodeGen.ini* . The first one is only loaded by the application, and it contains the list of component names. The other is loaded at startup and saved on close, and is used for general application settings. It is expected that both files are in the same directory as the executable.

If users want to install DynTFTCodeGen in a directory with restricted write permissions, like Program Files, the application would require to be started as Administrator, to be able to write to its *DynTFTCodeGen.ini* file. To avoid starting it as administrator, users can manually create a third ini file, called *DynTFTCodeGenIniOverride.ini* and place it in the same directory as the executable. This file should contain the path to *DynTFTCodeGen.ini*. Thus, *DynTFTCodeGen.ini* can be loaded from and saved to a directory without special write permissions.
In this case, *DynTFTCodeGenIniOverride.ini* should have the following structure:

```
[IniOverride]
Filename=Path\To\DynTFTCodeGen.ini
InstalledComponentsFilename=Path\To\DynTFTCodeGenInstalledComponents.ini
```

In the current version, the application does not save the *DynTFTCodeGenInstalledComponents.ini* file, but allows it to be loaded from a different location.

Schema files are found in the *\Schema* directory, relative to the executable. These files describe the available installed components. They contain the list of properties, events, constants and code templates for code generator. There is a common schema file (*BaseSchema.dynscm*), which contains properties, events and constants for all the other components. They roughly match the fields from the "TDynTFTBaseProperties" structure in *DynTFTypes.pas* file of the library.

Base properties from *BaseSchema.dynscm*, can be overridden in component schema files, for further customization. Schema files are also the place where properties are configured as run-time or design-time properties. As mentioned in the previous chapter, new components can be added by creating new Schema files and adding them to the *DynTFTCodeGenInstalledComponents.ini* file, but, they will neither have a drawing on the Drawing Board, nor an icon in the palette. Schema files are loaded once, at application startup.

OutputParts files are common sections of the generated files. They can be found in the *\OutputParts* directory near the executable. OutputParts files are loaded at every code generation. For now, the *DynTFTFonts.pas* file is completely generated without OutputParts files.

3. Component Palette

This part of the application's main window, displays the available components that can be worked with. Its content is described in *DynTFTCodeGenInstalledComponents.ini* file, and by default, it consists of 20 components, organized into two categories: "System Components" and "Other Components". Out of the 19 available system components, five are marked as "Do not use", because they are not designed to be standalone in a DynTFT project. They can be found by a red x icon next to them. Although unusable as standalone, they can be placed on the drawing board, for preview purposes. Except the DynTFTMessageBox component, if placed on the drawing board, all of these "Do not use" will cause the application to generate code in the output files, which should be perfectly fine.

In the "Other Components" category, there is a DynTFTComment component, which is configured not to generate code for the output files. It is a design-time component and it can be placed as much as needed on the drawing board. It can be configured to use custom fonts and colors.

To place a component on the drawing board, simply drag it from Component Palette. In case the component a user is looking for, is difficult to spot, there is a search box, below the list of components, which can be used to search for a component by its name.

4. Drawing Board

All components end up on the drawing board, to be placed into position and resized as needed. The drawing board allows components to be moved, resized, selected, copied, cut, pasted and deleted. Using the keyboard, components can be moved by holding the Ctrl key and pressing one of the arrow keys. For fast moving, also the Shift key has to be held, together with the Ctrl key. For resizing, only the Shift key has to be held, while pressing the arrow keys. As mentioned in the "Limitations" section from the Introduction chapter, components do not snap to a grid or to each other. There are however, alignment "guide" lines, which appear when one or more components are aligned to other components, during moving or resizing. They indicate top to top, left to left, right to right or bottom to bottom alignments.

The selection can be done by mouse, or by keyboard. When selected using the mouse, components from multiple screens can be added to selection, by holding either Ctrl or Shift keys while clicking on components. There is no difference between these keys when selecting. Both are

used identically for convenience. When done by keyboard, there are two shortcuts, Ctrl-A and Ctrl-Shift-A, to select all components. The first one is used to select components from the current screen, while the other selects all components across all screens.

On the drawing board, components have a pop-up menu, which allows cutting, copying, deleting and bringing them to front or sending them to back. The drawing board itself has its own pop-up menu, which allows pasting components from clipboard and selecting all components. The copy-paste operation uses a text clipboard format, which allows these operations across multiple instances of the application. As a restriction, the drawing board has to be focused when copying using the Ctrl-C keyboard shortcut. The application is configured by default to focus the drawing board when hovered with the mouse. Unfortunately, there is no indication when it is focused or not.

Also from the drawing board, the screen size is configured, by dragging the two red bars. For fine tuning, these screen edges can be moved using the arrow keys on the keyboard, while holding them with the left mouse button. For now, their color is not configurable and they can't be hidden. However, they can be locked using their pop-up menu. The screen size, configured by these two bars, does not directly configure the screen size in a DynTFT project. They have to be manually matched. The screen size is set by default to 480x272 and this is a project-level setting.

5. Object Inspector

When one or more components are selected, the object inspector is populated with component properties, based on the current selection. This is where component properties and event handlers can be set. Properties are organized into two categories, design time and run time. In both sections, there are properties which come from the base schema file and properties which are component related. Some property names are hardcoded into application and expected to exist. These include "ObjectName", "ScreenIndex", "Left", "Top", "Width", "Height", "CreatedAtStartup", "HasVariableInGUIObjects", "Locked", "ActiveFont", "MinWidth", "MinHeight", "MaxWidth", "MaxHeight", "Count", "AllButtonWidths", "AllButtonLefts", "PageCount" etc.

5.1 Object Inspector overview

The run time properties, match the available fields in a component's data type structure, by name. More than that, there are run time properties, which match fields from subcomponents. This is to allow code generation by property name, even for subcomponents. As a downside, selecting two components, with a common property, where one is a property of a subcomponent, will end up hiding both, because of the name prefix. For example, the DynTFTItems component has an "ActiveFont" property. This is also displayed on a DynTFTListsBox as "Items^.ActiveFont". Because of the "Items^." prefix, when selecting both components, these properties will not be displayed on the object inspector.

The design time properties are the ones which are not part of the component's data type structure, but control the behavior while designing and generating code.

When placing a new component on the drawing board, its properties are initialized to their default values, as defined in the schema file of that component. If a run time property is modified by user, then assignment code for that property is going to be generated. This is also the case for indirectly modified read-only properties (see below).

Some of the property values are validated at design time, to avoid generating uncompilable code. Others are left unvalidated on purpose, to allow the freedom of assigning everything the user wants. Based on their datatype and name, some properties have different local editors (Edit, ComboBox, ColorBox). For those which use combo boxes or color boxes, they come preloaded with a list of available constants. Unfortunately, these constants are not grouped, like values of an enumeration, so all of them will be displayed in the same list.

Although useful, the application does not validate string length or number of items of a property. When a string property contains a value, longer than what the field of the DynTFT component data structure allows, it will generate a compilation warning, like: "String constant truncated to fit STRING[19]". Users will have to keep track of these warnings and prevent such cases, to not allow memory corruption on the microcontroller application.

5.2 Common properties

The "ObjectName" property defines the name of the DynTFT component at code generation and has to be unique, because all of these names will become variables, part of the same namespace. The variables, pointing to the actual DynTFT components at run time, are generated in the *DynTFTGUIObjects.pas* file. For components, which have no properties to be set in the DynTFT project at run time, by the user code, there is an option for skipping generating a variable in the *DynTFTGUIObjects.pas* file. This is controlled by the "HasVariableInGUIObjects" property. When set to False, a local variable is generated in the associated "CreateGUI_Screen" procedure from the DynTFTGUI.pas file. Most of the times, this results in smaller flash and RAM usage for the microcontroller application. If no property is set for such a component, none of its fields will be set in the "CreateGUI_Screen" procedure, so a variable will be generated and cause a compiler notification, for being assigned but not used further.

When the "CreatedAtStartup" property of a component, is False, no code is generated for that component, although present on the drawing board. This is to allow users to create that component manually when needed and preview how it will look like. As a limitation, the initialization / component content code is still generated in *DynTFTGUI.pas* file, and not available in *DynTFTGUIHandlers.pas*.

The "Locked" property controls the editing of position and size of a component, at design time. When set to True, the component can't be moved or resized. However, the component can be cut and pasted to a different location even when locked.

The "ActiveFont" property allows setting a different font to a component. The "ActiveFont" field of a DynTFT component data structure, is available only when the "DynTFTFontSupport" compiler directive is present in the DynTFT project, and only for components which display text. Because it depends on compiler directives, the property name is displayed in italic and has the "<with directive>" suffix in object inspector. Also, its value is displayed in red. By default, it is set to "@TFT_defaultFont". To set a new font, it must be first created in the Project Settings dialog (see "Tools" item of the main menu). After creating the new font, it will be available in the selection list of the "ActiveFont" property in object inspector. Notice the "@" used in front of the font constant name. This is because the generated code contains this value, unmodified. When the "UseExternalFont" compiler directive is defined in a DynTFT project, the "TFT_Set_Ext_Font" function is used, so the value of "ActiveFont" properties might have to be set to a value without "@". There is no automatic switching between these two options and DynTFTCodeGen does not keep track of the "UseExternalFont" compiler directive. This property might be present with a suffix in object inspector, if it is part of a subcomponent.

Using the "MinWidth", "MinHeight", "MaxWidth", "MaxHeight" properties, the size of a component can be constrained at design time. When these properties are set to 0, they have no effect.

Other properties like "Count", "AllButtonWidths", "AllButtonLefts", "PageCount", are component specific and are modified/updated by the application, either at design time or at code generation. They are set to read-only in their schema files, to avoid being used at design-time. For example, the "AllButtonWidths" property is a list of integers with the "Width" values of all buttons from a PageControl. It is internally updated, based on the component configuration. Properties like these are required at code generation when there is custom initialization code in the schema files.

5.3 Editing event handlers

Event handlers can be edited from the second tab of the object inspector. By double clicking in the "Value" column, for a particular event, a new event handler name will be generated. Based on their datatype (handler header definition), multiple events can be assigned to the same handler. The available list, when adding or modifying a handler may contain handler names, which are not assigned. They will be discarded when closing the project. However, when generating the output files, these unassigned handlers will also be generated and will have to be manually removed if not needed. They are not automatically removed, because users may want to assign them manually for dynamically created components.

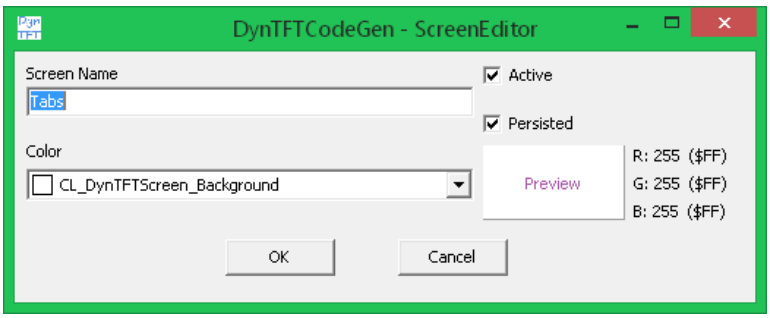
6. Screens

As mentioned before, components are organized into screens, so they won't be displayed all at once. Screens can be added, removed and edited either from the pop-up menu of the list of screens (the tab buttons above the drawing board on the main window) or the "Edit"->"Screen" items of the main menu. Also from here, screens can be set to Active/Inactive and "Persisted/Not persisted. A component, belonging to a screen can be easily moved to another screen, by modifying its "ScreenIndex" property in object inspector. This is available at design time only for now.

The active/inactive setting of a screen is used at application startup in a DynTFT project, as an initialization value for the "Active" field of the "DynTFTAllComponentsContainer" array (see "SetScreenActivity" procedure in a *DynTFTGUI.pas* file).

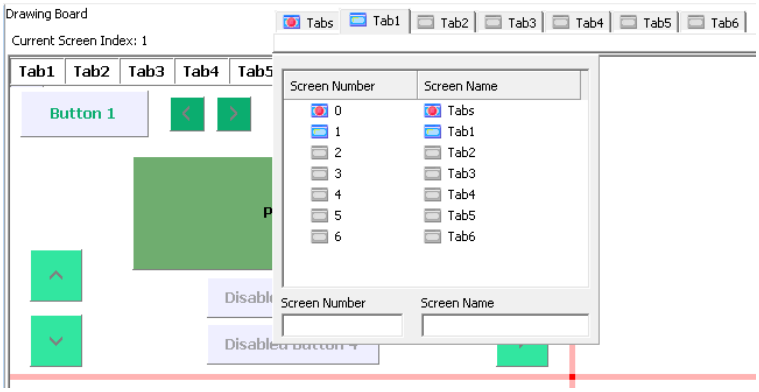
The "Persisted" setting allows displaying components of one screen when other screen is currently selected. This is a design time setting only.

Screens can have a background color and be set to draw a "clear screen" rectangle at application startup. In addition to the "Active" and "Persisted" settings, the screen editor, displayed in the following screenshot, also allows setting the screen color. This screen color is also used at run time when dynamically deleting components or repainting different areas of the screen. The color can be set to one of the predefined colors or to a DynTFT color constant. Setting to a DynTFT color constant, the screen color is automatically updated when a color theme is used.



For applications with many screens, DynTFTCodeGen allows searching for a particular screen by its name. This is available either from the pop-up menu of the screen, or the main menu. By clicking the "Search for screen..." item, a small pop-up dialog opens, with a list of screens and two search boxes (see the following screenshot). Users can type in either the screen number or the screen name. To go to the desired searched screen, just double click on one of the items in the list. This will change the current screen and dismiss the dialog.

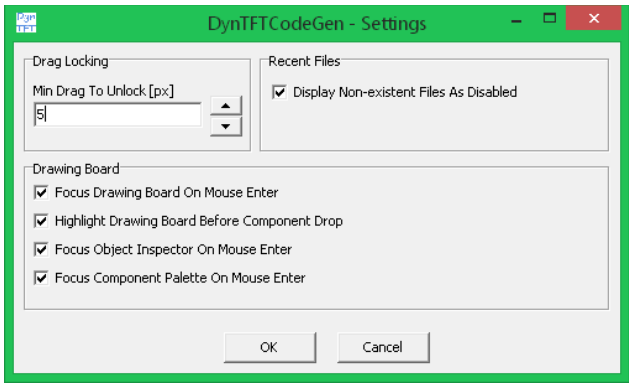
The screen name is not part of a namespace, it is simply a design convenience, so there is no restriction in having two or more screens with the same name.



There are also few screen related settings on project settings dialog, which will be discussed later, in another chapter.

7. Application level settings

Accessing general application settings can be done from the main menu, "Tools" -> "Application Settings...". A small dialog opens, as in the following screenshot:



The "Min Drag To Unlock [px]" editbox allows setting the minimum number of pixels the mouse has to move when dragging a component, to actually start the moving operation. The default value is 5 pixels and this feature can be switched off by setting it to 0. When set to 0, a component is dragged by mouse immediately as the mouse moves. This setting prevents accidental dragging of a component when clicked. Its maximum value is restricted to 10px. If that is still too small for a very sensitive mouse, the components can be locked.

The list of recent files, under the main menu, "File"->"Open recent project" item, contains all files loaded or saved by the application. Over time, some of them may be deleted, renamed or moved on disk, so they can be displayed as disabled, using the "Display Non-existent Files As Disabled" checkbox.

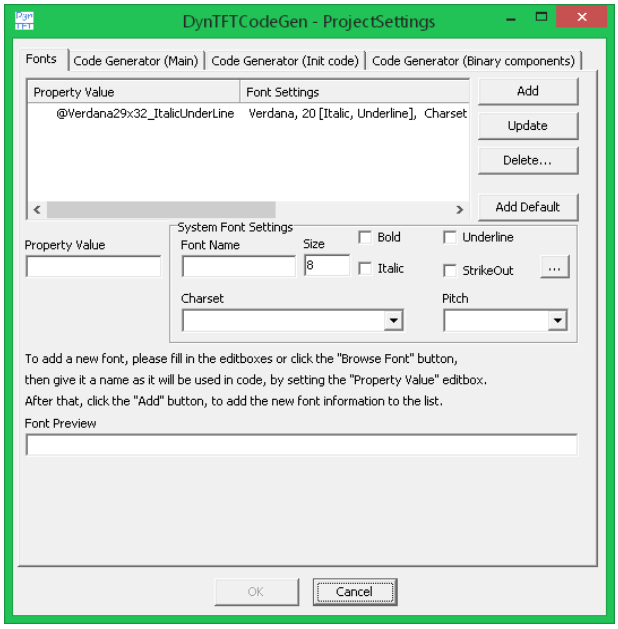
For the drawing board, there are currently four options, under the "Drawing Board" group box. There are three "Focus.." checkboxes, for the drawing board, object inspector and component palette. When checked, one of the three mentioned components is automatically focused when hovered by mouse. This is especially useful for the drawing board, because various operations like Copy/Cut-Paste, moving components, resizing components, selecting component etc, are active only when the drawing board is focused. When another component is focused, the drawing board loses focus, so its keyboard shortcuts won't be available. When the "Focus Drawing Board On Mouse Enter" checkbox is unchecked, the drawing board will have to be focused manually by a mouse click. The other two "Focus.." options are mainly for causing the drawing board to lose focus.

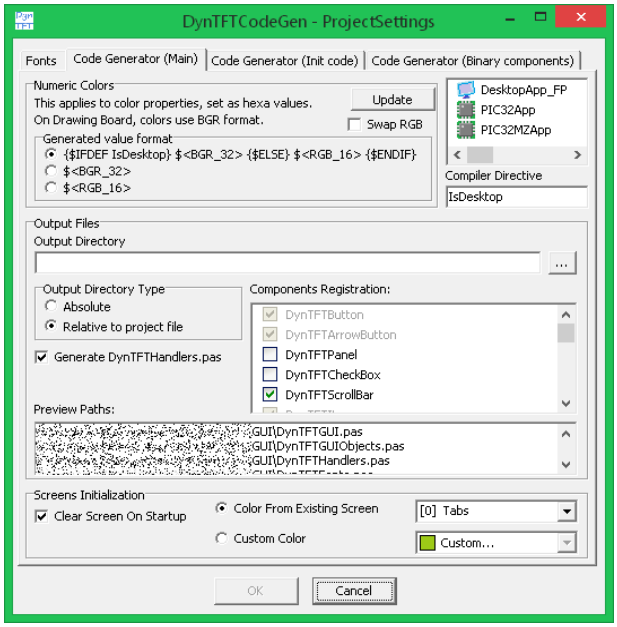
When dragging a component from the component palette to the drawing board, the drawing board can be highlighted with an extra border. This can be switched off by unchecking the "Highlight Drawing Board Before Component Drop" checkbox.

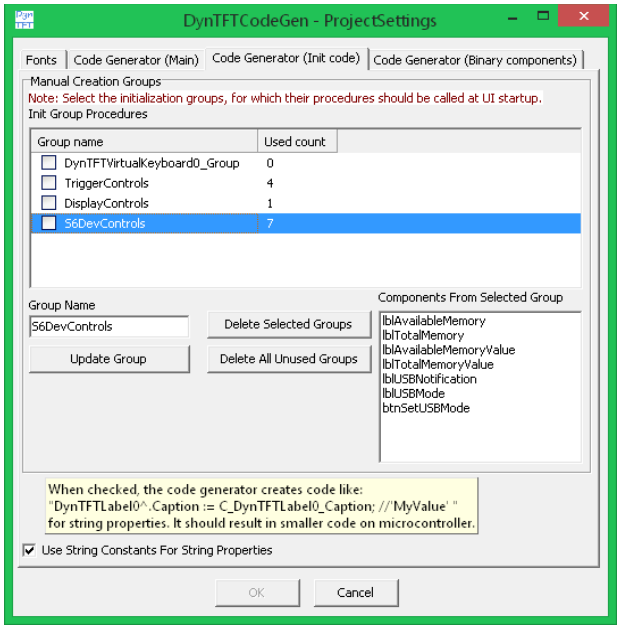
These general application settings are saved to the *DynTFTCodeGen.ini* file when closing the application.

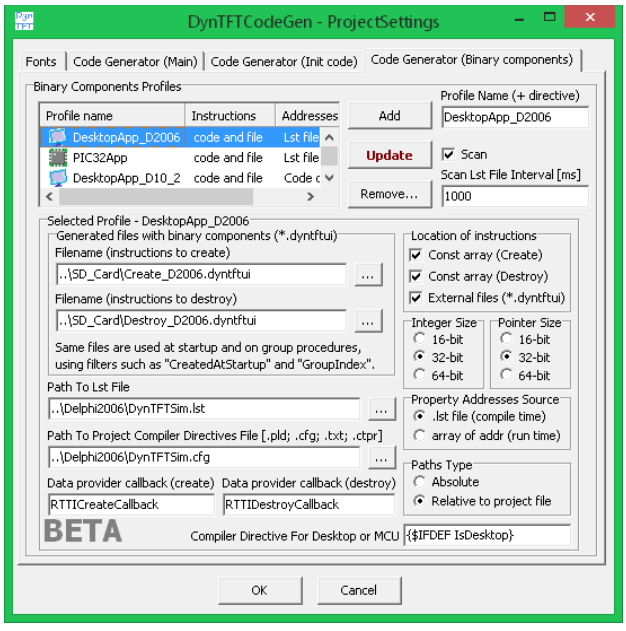
8. Project level settings

Every project can be customized with a multitude of settings, from the "Project Settings" dialog, accessible from the main menu, "Tools" -> "Project Settings...". See the following screenshots:









The dialog consists of multiple pages, "Fonts", "Code Generator (Main) / (Init code) / (Binary components)". Other settings, like the screen settings, are not included in this dialog.

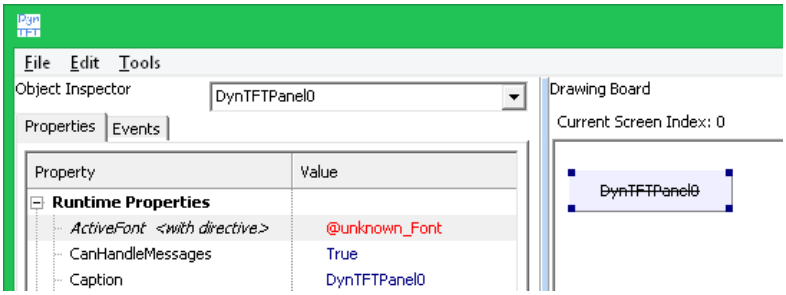
8.1 Font settings

From the first page of the dialog, fonts can be added, edited and removed from a project. They have to exist in a project prior to being used as values for the "ActiveFont" property of various DynTFT components.

To add a new font, either browse system fonts, using the "..." button, under the "System Font Setting" groupbox, or manually type a font name in the "Font Name" editbox. Set its options under the same groupbox, then give it a name in the "Property Value" editbox. This name will appear as an option when setting the "ActiveFont" property of a component. It can also be preceded by the "@" character if fonts will be included in the DynTFT project as internal fonts. After filling in all these settings, press the "Add" button. The new font should be added to the list. As an example, the default font can be added from the "Add Default" button. This will add "@TFT_defaultFont" as Tahoma, 10, bold. It has to be mentioned that adding the default font to a project is not needed, because it automatically exists and is used by default. However, if added and customized, all components which use it, will display the new font settings. This is useful when using custom TFT libraries, which either hardcode the default font settings or allow the font to be customized.

Updating an existing font, requires the font to be selected in the list, its new changes to be set, from the "System Font Setting" groupbox, then the "Update" button to be pressed. Deleting a font can be done from the "Delete..." button, if selected in the list.

When selecting a font which does not exist, as a value for the "ActiveFont" property, the application will display a striked out small font. See the following screenshot:



This can happen either after removing a font from the list, or typing in a non-existent font.

As mentioned before, switching a DynTFT project from internal to external fonts, using the "UseExternalFont" compiler directive, may require removing the "@" prefix from font property values. There is no automatic means of doing this, so font settings will have to be updated manually. The same for components.

8.2 Code generator (Main) settings

On the "Code Generator (Main)" tab, from the "Project Settings" dialog, various options are available. There are three group boxes, "Numeric Colors", "Output Files" and "Screens Initialization".

The "Color" and "Font_Color" properties of DynTFT component, can be assigned to color constants or their numeric counterpart, in hexa format. DynTFTCodeGen works with 32-bit colors (the most significant byte is not used), in BGR format, as provided by the operating system. Color constants are defined in the DynTFTConsts unit, for the desktop simulators and the TFT library for the microcontroller applications. In addition to that, theme specific colors are defined in their color themes as well. All these constants are declared in two ways, 32-bit (BGR) and 16-bit (RGB). DynTFTCodeGen can generate code for assigning "Color" and "Font_Color" properties with any of these options. The "Generated value format" group box allows selecting between a switch of 32-bit vs. 16-bit values for desktop vs. MCU code, 32-bit only or 16-bit only. When selecting the first option, "{$IFDEF IsDesktop} $<BGR_32> {$ELSE} $<RGB_16> {$ENDIF}", the generated code

contains both versions of the color, in 32-bit and 16-bit format. For desktop, it will use 32-bit and for the microcontroller, the 16-bit one, as specified by the "IsDesktop" compiler directive. This compiler directive can be changed from the "Compiler Directive" edit box, but for now, this is the name the DynTFT library supports. It has to be mentioned that the same compiler directive is used when generating code for the "ActiveFont" property, between desktop simulator and MCU project.

The second option of the "Generated value format" groupbox configures the code generator to generate only 32-bit values, (BGR). This is desired when using custom libraries, which support 32-bit colors. The third option will generate 16-bit only values in RGB format. The last setting in this group box is the "Swap RGB", which causes the code generator to generate all color values with red and blue channels being swapped. This may also be the case for custom TFT libraries.

When using binary components, each settings profile will have a different set of color formats. To update a profile, select it in the list, set a format, then click the "Update" button.

The "Output Files" group box, allows setting the output directory for the four output files, *DynTFTGUI.pas*, *DynTFTGUIObjects.pas*, *DynTFTHandlers.pas* and *DynTFTFonts.pas*. The *DynTFTFonts.pas* is used for the DynTFT simulators only. The output directory can be relative to the project file, or it can have an absolute path. For relative path, the application can't properly handle network locations. According to the path settings, the "Preview Path" listbox displays the actual path of the output files. It displays the path only when the DynTFTCodeGen project is saved.

By unchecking the "Generate DynTFTHandlers.pas" checkbox, the *DynTFTHandlers.pas* will not be saved/modified by DynTFTCodeGen.

The last setting in the "Output Files" group box, is the "Components Registration". It is used when generating the "RegisterAllComponentsEvents" procedure from the *DynTFTGUI.pas* file. Every registered component increases code size in the final microcontroller application, so commenting out unused registration function, helps decrease the size. DynTFTCodeGen automatically sets various components to "checked", in this list, if they are used in the project or they are dependencies of used components. Users can additionally check components if they know they will be created dynamically at run time, instead of being handled by DynTFTCodeGen. When a simulator displays an exception like `Exception while executing DynTFT_GUI_LoopIteration: PDynTFTMessageBox was not registered. Please call RegisterMessageBoxEvents before creating a PDynTFTMessageBox. It should be called once in RegisterAllComponentsEvents procedure from DynTFTGUI unit.`, then a component registration is missing, which can be added from the "Components Registration" list.

From the last group box, "Screens Initialization", users can choose to clear the screen with a rectangle, at application startup. This can be done by checking the "Clear Screen On Startup" checkbox. The "clear screen" color is selected from the remaining controls. When the "Color From Existing Screen" radio button is selected, the color used comes from one of the available screens. For the "Custom Color" option, users can use any of the available color constants, or a custom color.

8.3 Code generator (Init code) settings

The code generator can add special code for creating and destroying components at runtime, under user control. This is done by grouping various components together under "creation groups", mentioned by their "ManualCreationGroup" property. Users just have to type in a new or existing group name in that field and set the "CreatedAtStartup" property to False. For these components, the user can call their creation or destruction code when needed, not at startup as for the others. This code is placed under "CreateGUIGrp_<GroupName>" and "DestroyGUIGrp_<GroupName>" procedures, in *DynTFTHandlers.pas* unit. These procedures can be called, for example when changing the active screen, to destroy some existing components and create new ones.

The settings under the "Code Generator (Init code)" tab from the Project Settings window, allow editing group names and even generate code for calling the creation code at UI startup.

8.4 Code generator (Binary components) settings

For a better understanding of the "binary components" set of settings of the code generator, see chapter 10. This section is mostly about the settings available on the dialogbox.

Every targeted compiler is required to have one settings profile, to use RTTI instructions. This is required, because every project can have its own set of compiler directives, used as feature toggles in DynTFT library, and also because event handlers will have different memory addresses across projects/compilers/architectures

The upper section of the tab is a list of settings profiles, its add/update/remove buttons, a profile name editbox and two timer settings for scanning .lst files. The list of profile names displays all available profiles in the current project. By clicking on one profile from the list, its settings are displayed. The profile name (see "Profile Name (+ directive)" editbox) has to follow syntax identifier rules, because it is also used as compiler directive for profile related code. The "Scan Lst File Interval [ms]" editbox specifies the interval used when scanning lst files for modifications. The lst scan timer is active when the "Scan" checkbox is checked. There is only one timer for all profiles. Except these two timer settings, all the other settings on the " Binary components" tab, belong to a profile.

Adding new profiles and updating existing ones, using the "Add" and "Update" buttons, will add/update them, using the current settings from the "Selected Profile - <ProfileName>" groupbox. Click on Add button to add the current settings as a new profile. Click on Update button to update the selected profile with the current settings. Click on Remove, to delete the selected profile.

Under the "Selected Profile - <ProfileName>" groupbox, there are settings for generated output files: "Generated files with binary components (*.dyntftui)" groupbox, input data: "Location of instructions" groupbox, data type bitness: "Integer Size" and "Pointer Size" groupboxes, property addresses location: "Property Addresses Source" groupbox, paths settings: "Paths Type" groupbox, path to lst file, path to compiler directives file, name of data provider callbacks and a compiler directive setting editbox.

- Use the "Filename (instructions to create)" editbox, to specify the .dyntftui file, which will be generated by DynTFTCodeGen, with RTTI instructions to create and set components.

- Use the "Filename (instructions to destroy)" editbox, to specify the .dyntftui file, which will be generated by DynTFTCodeGen, containing RTTI instructions to destroy components.

- From the "Location of instructions" groupbox, users can specify how the instructions will be stored. There are two available options: in code, and in .dyntftui files. When using code, instructions are stored as two constant arrays, one for creating components / setting properties, and the other for destroying components. For projects, which do not have manual creation groups, i.e. components are created once, at startup, there is no need to generate destroy instructions, so the "Const array (Destroy)" checkbox may be left unchecked. It is however, required for the "Const array (Create)" checkbox to be checked, to generate create/set instructions in code. When checking the "External files (*.dyntftui)" checkbox, two .dyntftui files are generated, one for creating/setting components, and the other for destroying components. Although the .dyntftui files are created, they won't be used, unless callbacks (one for each file, create and destroy) are provided in code, and their names configured into the "Data provider callback (create)" and "Data provider callback (destroy)" editboxes. Different files have to be configured across settings profiles, to avoid overwriting them.

- Use the "Integer Size" and "Pointer Size" groupboxes to configure the bitness of the targeted architecture. Each of the two groupboxes, has three available options, 16-bit, 32-bit, 64-bit. For example, Integer data type is 16-bit on PIC24/dsPIC and PIC32, and is 32-bit on Desktop x86. On Desktop x64, Integer would be 64-bit. The pointer size does not have have to match Integer data size. It can be 16-bit or 32-bit on PIC24/dsPIC, depending on how the "TPtr" data type is defined in DynTFTTypes.pas from the DynTFT library, it is 32-bit on PIC32 and Desktop x86, and is 64-bit on Desktop x64. Proper configuration of these two datasizes is required, for each profile, because this

directly affects settings properties of Integer, Boolean and PByte (pointer) types. Boolean properties are assumed to be of pointer size, except on x64, where they are used as 32-bit. Because of this, properties of Boolean type are defined as LongBool in DynTFT components. If not configured properly, data can be truncated or undesired memory locations will be overwritten (i.e. memory corruption) when setting properties, during RTTI instruction execution.

- Event handlers and fonts, used as PByte type (and maybe other future data types) can be encoded either directly, specifying their addresses (obtained after compiling and linking), or by placing them into an array, and indexing that array at runtime. The "Property Addresses Source" groupbox has two available options for this. One is to read the addresses from the generated .lst file and the other is to generate an array of event handlers and fonts, as part of the code.

When using the .lst file, event handler and font addresses are written to the .dyntftui files or the generated constant array of instructions (not to be confused with the above mentioned array of addresses), as they are found in the .lst file. It has to be mentioned that there may be multiple compile/generate cycles needed to properly synchronize these contents, because DynTFTCodeGen does not have at every code generation, all the required addresses. These would be available only after generating the .lst file, which requires all source files to be generated (chicken and the egg problem). Nonetheless, multiple compilations of the same project, will eventually result in the same placement of the event handlers and fonts, i.e. same .lst file content across two consecutive compilations. The inconvenience caused by multiple compilations can be payed off by the fact that this option results in the smallest code size among all the available settings. It has to be mentioned that the desktop simulators must also generate a .lst file if using this option. The DynTFT library provides the *DynTFTListExporter.pas* unit, and DynTFTCodeGen automatically generates code required by this unit. The downside is that on desktop, addresses are not available at compile time. Because of this, the simulator has to be started, to generate/update its .lst file.

The option to generate code with all event handlers and fonts into an array, and read their addresses at runtime is a bit more convenient, because there is no need to synchronize the generated files with the resulted .lst file, i.e. no need for multiple compilations to get the addresses right. This option has an increased code size, compared to the .lst file option, because of the array of addresses and the required code to read it. The decision to use hardcoded addresses, or to index handlers and fonts, is made by the RTTI execution engine, as this information is encoded in each "set property" instruction. However, DynTFTCodeGen generates only one of the two options at a time.

- Using the "Path To Project Compiler Directives File [.pld; .cfg; .txt; .ctpr]" editbox, each settings profile can specify what list of compiler directives to use. This is required, because various component properties depend on compiler directives (e.g. the "Items^.ItemsEnabled" or the "Items^.OnGetItem" properties of a ListBox). Even fonts depend on "ActiveFont" property of various components and also on "DynTFTFontSupport" compiler directive. Since these options can be configured differently across projects, DynTFTCodeGen has to "preprocess" them, to properly generate instructions only for the available properties. Another requirement of these lists of compiler directives, is to make sure the profile name, as set in the "Profile Name (+ directive)" editbox, is also defined as a compiler directive.

- All of the four mentioned paths, specified on this tab, can be configured as absolute paths or relative to the project file. This is done from the "Paths Type" groupbox. When using relative paths, the project has to be saved, so that DynTFTCodeGen would know what project filename to use. Network paths may not be supported as relative paths.

- Every profile will also need to specify if it targets Desktop or MCU, using the "Compiler Directive For Desktop or MCU" editbox. Right-click on this editbox for suggestions.

- As mentioned before, to use .dyntftui files, the names of two callbacks have to be specified to the "Data provider callback (create)" and "Data provider callback (destroy)" editboxes.

More information about the above settings, can be found in component tooltips. Just open the dialog and hover every component with the mouse.

9. Code generation

Many features of the code generator were mentioned in previous chapters, leaving here the most specific ones. The code generator is able to fully generate all four files, *DynTFTGUI.pas*, *DynTFTGUIObjects.pas*, *DynTFTHandlers.pas* and *DynTFTFonts.pas* and to edit the already existing *DynTFTHandlers.pas* file. The first three files are included in a DynTFT microcontroller project, via the *DynTFTGUI.mpas*, *DynTFTGUIObjects.mpas*, *DynTFTHandlers.mpas* files. The last one, *DynTFTFonts.pas* does not have a microcontroller counterpart, because it is needed at desktop only.

Editing the *DynTFTHandlers.pas* file means adding event handlers, both in the interface section as the headers, and in the implementation, their implementation code. To identify various sections of the file, and allow proper code inserting, the application uses "code generator symbols". These are strings, added to the headers of procedures/functions and also to their "begin" and "end" keywords. They come as comments at the end of a line, and look like "`//CodegenSym:header`", "`//CodegenSym:handler`", "`//CodegenSym:handler:begin`" and "`//CodegenSym:handler:end`". Please do not remove these strings, to allow proper application functioning!

As mentioned before, DynTFTCodeGen does not keep track of various DynTFT project settings, defined by compiler directives, so they will have to be manually managed. Parts of the generated files come from the "OutputParts" files from the *\OutputParts* directory, allowing future project customizations. Users can even install DynTFTCodeGen into separate directories with different "OutputParts" files, depending on application needs. For most of the cases, include files (*.inc) can be used to add custom units and raw code.

Custom TFT libraries can be used when the DynTFT project defines the "UserTFTCommands" compiler directve. As mentioned before, color themes have to be manually configured, DynTFTCodeGen allowing only to preview them.

10. RTTI and binary representation of UIs

DynTFT features a runtime type information mechanism for design components, which allows loading of UIs from an external medium. Type information is kept to a minimum, because of memory limitations in microcontrollers. It includes component registration index, which identifies used component types within a design, and component property regsitration index, which identifies properties within a component type. Using type information, UI descriptions can be binary encoded for both better compression of initialization code, and storing UIs on external media.

10.1 Overview of RTTI and binary components

Storing UIs in binary format can be done either as constant arrays in code, or as two files on an external medium. Such an UI is represented as a stream of instructions, used to create or destroy components and to set component properties. To recreate an UI from this content, a small execution engine is called at application startup and/or on demand (user code), which can create components and set their instructions. Also on demand, it can destroy components (see manual creation groups). Throughout the DynTFTCodeGen application, these instructions are called "RTTI instructions", because they depend on type information to work with components in a generic way.

DynTFTCodeGen is able to generate mixed UIs, i.e. containing old fashioned code (e.g. `MyComponent := PMyDynTFTComponentType.Create(<arg>..<arg>);`) and similar information, encoded as binary. The encoding granularity is at component level, by setting the boolean property, "UseBinaryEncoding" for desired components.

Working with RTTI and binary encoded UIs, requires defining the *RTTIREG* compiler directive in a DynTFT project. The generated code uses this directive for backward compatibility.

"Binary components", i.e. those which have the "UseBinaryEncoding" property set to True, will not be created in a project which does not define the *RTTIREG* compiler directive. Another similar requirement is to define a settings profile for each DynTFT project, which uses RTTI. This can be done from the project settings dialog, "Code Generator (Binary components)" tab. Without a settings profile, components marked as binary, will not revert to the non-RTTI creation code.

The limitations of binary encoded instructions come from DynTFTCodeGen, the DynTFT library itself, programming language compatibility and support, speed and size optimizations and also from hardware.

Here are some of the limitations of binary encoded UIs:

- maximum number of components: 32767 (limited by indexing with 16-bit integer)
- maximum number of component types: 128 (limited by instruction encoding)
- maximum number of properties / component type: 128 (limited by instruction encoding)
- maximum number of bytes in a string: 255 (limited by instruction encoding and constrained strings on desktop application – library limitation)
- minum buffer size on loading content from file: 256 bytes (required by instruction encoding and setting string properties)
- maximum number of manual creation groups: 31 (limited by instruction encoding)
- maximum number of event handlers: $2^{31} – 1$ (further limited by compiler and memory)

Also as a limitation, a generated binary UI, has to match the runtime registration code (see "RegisterAllComponentsEvents" procedure from *DynTFTGUI.pas*, which is generated by DynTFTCodeGen). Similar to this, are component properties and event handlers. It has to be mentioned that event handlers are not binary encoded. Their code/content is kept as before, but they are simply assigned to event properties, using binary encoded information. Since event handlers are special type of properties, they can be encoded either as an index of an actual event handler address from an array of addresses, or can be encoded as the address itself when available. Like event handlers, font addresses are also encoded in the same way.

Another type of limitation comes from various combination of code generation options, like manual creation groups and the "CreatedAtStartup" and "HasVariableInGUIObjects" properties of a component. When using manual creation groups, the RTTI execution engine is called from the "group procedures" in addition to the startup code, to allow creating and destroying components like before. Components which are part of a creation group and also have both the "CreatedAtStartup" and "HasVariableInGUIObjects" properties set to False, will not be created. A component, which does not have a variable to hold its object, if created, will not be able to be destroyed, because there is no variable to point to that object. Components, which have the "HasVariableInGUIObjects" property set to False and "CreatedAtStartup" is set to True, are expected to be created and never destroyed. Regardless of these settings, these components will be part of the generated instructions, which can be either executed or ignored.

Using RTTI, is more of a memory and code size overhead than processing power, but nonetheless, it provides smaller code size of the same UI and also the possiblility of storing it on external media. For big UIs (i.e. many components and design-time assigned property values), the RTTI engine is smaller that the UI itself, resulting in an overall decrease of code size, compared to a non binary UI. For smaller UIs, there is no benefit of using RTTI.

10.2 Location of instructions and Property Addresses Source

As also mentioned in the description of the settings dialog, RTTI may use different arrays in code, depending on selected settings. Instructions can be generated as .dyntftui files or can be stored in two constant arrays (see `C_RTTI_CreateInstructionData` and `C_RTTI_DestroyInstructionData` constants in a generated *DynTFTGUIObjects.pas* file). Another array, mentioned accross this

documentation is a variable (see `AllBinHandlers` in a generated *DynTFTHandlers.pas* file), containing the addresses of event handlers and fonts, to be read at runtime by the RTTI execution engine. The `AllBinHandlers` array is used only when the "Property Addresses Source" groupbox is set to "array of addr (run time)". In this case, each "set property" instructions, which targets an event handler or font, will contain the index of that event hander or font in the `AllBinHandlers` array. This option is available for both the .dyntftui files and the constant array of instructions for creating and setting components, `C_RTTI_CreateInstructionData`.

Using the generated .lst files, to get the addresses of event handlers and fonts, is the most efficient encoding when looking at final code size, but it is the worst compile time option, because it requires multiple compilations and manual monitoring of the generated .lst files.

The smallest code size option is to use dyntftui files, stored on an external memory, rather than using a constant array in code. This is true only for the case, where the application already uses that external memory (SD card, EEPROM, Flash etc.) for other purposes. The main downside of using .dyntftui files, is that the external memory has to be connected (e.g. an SD card has to be present) when executing instructions. Also to be mentioned, is the fact that on MCU, if using short filenames (although "dyntftui" is a long extension), results in smaller code size (maybe it matters).

As a limitation, DynTFTCodeGen can generate only one set of instructions per project, i.e. only one UI per project. This means that an application, which requires multiple UIs to be changed at runtime, must have multiple projects to generate all the necessary sets of .dyntftui files. However, it would be easier to group components into manual creation groups and control their creation and destruction at runtime, while using the same two .dyntftui files.

It has to be mentioned that DynTFTCodeGen does not implement an automation feature for copying the newly generated .dyntftui files to the storage media (e.g. SD card).

These settings represent a tradeoff between code size, speed and compile time.

10.3 Using lst files

The option of reading event handlers and fonts addresses from .lst files, is the most demanding when it comes to user effort. This is because multiple compilation / code generation cycles are needed, to properly update the RTTI instructions (either code constant or .dyntftui files), to match the addresses of the compiled event handlers and fonts. The effort comes from the fact that on every UI change, which involves a regeneration of output files (.pas files and/or .dyntftui files), the projects across all targeted architectures, have to be rebuilt. As a result of rebuilding the projects (also starting the desktop simulators), their .lst files are updated with the proper addresses of event handlers and fonts, but most likely, they do not match the addresses, which DynTFTCodeGen used to generate the instructions. This is displayed both on DynTFTCodeGen's status bar and its messages dialog (can be shown from main menu -> Tools -> Display Code Generation Messages), as mismatched addresses. If this is the case, the output files (.pas files and/or .dyntftui files) have to be regenerated, to contain the latest addresses. Regenerating these files, requires another rebuild of the projects (only some projects may report mismatching addresses). Rebuilding the projects may or may not result in placing the event addresses and fonts at different addresses than before. At this point, if they are placed at new adresses (again), either multiple rebuilds (and starting the simulators) would be needed, or another code generation would be required to adapt to the new reported addresses. DynTFTCodeGen does not implement an automation mechanism to run the compilers, based on the reported mismatched addresses.

As mentioned in the description of the project settings dialog, DynTFTCodeGen implements a .lst scanning feature, using a timer. This timer can be configured from the setting dialog and should be enabled to allow looking for .lst file changes. It works by keeping track of the .lst files timestamps, across settings profiles and it opens these files only when their timestamps are changed (i.e. files are updated). The timer stops when closing a project and restarts if opening a project,

which is set to run the timer.

Users have to make sure that after generating the output files (.pas files and/or .dyntftui files), DynTFTCodeGen may report that all addresses are the same across its in-memory content and the verified lst file(s). However, a project rebuild is mandatory (and also running the desktop simulator(s)), to make sure the new changes will take effect, i.e. the existing lst file, used for creating the RTTI instructions, will be updated accordingly to the executable (.hex / .exe). Rebuilding/regenerating files have to be done until there are no more reported mismatching addresses in the messages dialog / status bar.

As already mentioned above multiple times, the desktop simulators have to be started, to update their .lst files, in addition to being compiled. This is because on desktop, addresses can't be properly obtained at compile time, as opposed to MCU. A `GenerateListFile` call has to exist on every desktop simulator, which has to generate a .lst file, and also to include the provided *DynTFTListExporter.pas* from the DynTFT library.

When the option of using .lst files, is desired, it is recommended to work on the design using another option (like placing addresses in the `AllBinHandlers` array), then switching to the .lst option from time to time, to see how it performs.

10.4 Setting compiler directives

To use RTTI instructions, multiple compiler directives are required. Some of them have to exist in all projects, others only particular to a project. For example, to enable the RTTI feature in the DynTFT library, the *RTTIREG* compiler directive has to be defined at project level. DynTFTCodeGen verifies if this directive is defined in the configured projects and reports an error if not found. As a limitation, it verifies only when generating files.

From the project settings dialog, settings profile names can be configured. These names are also used as compiler directives, particular to a project. This is required, because *DynTFTGUI.pas*, *DynTFTGUIObjects.pas*, *DynTFTHandlers.pas*, and *DynTFTFonts.pas*, are common for all projects in a DynTFTCodeGen design. The RTTI related code from these files is generated in multiple "copies", enclosed by profile names as compiler directives. They have to be unique, to properly compile. As with the *RTTIREG* directive, DynTFTCodeGen also verifies the existence of the profile names as compiler directives. At least for Delphi, adding compiler directives to the project, has to be made from the IDE, not by directly editing the .cfg file. This is because DynTFTCodeGen verifies .cfg files only, while Delphi may rely on project files (.bdsproj / .dproj).

Another set of compiler directives is configured from the "Compiler Directive For Desktop or MCU" editbox from the project settings dialog. Every profile has to specify, using this directive, if targets desktop or MCU. Two values are supported for now, "{$IFDEF IsDesktop}" and "{$IFDEF IsMCU}".

10.5 Miscellaneous

As mentioned before, to use .dyntftui files, two callback procedures must be implemented, and their names have to be specified in the project settings dialog ("Data provider callback (create)" and "Data provider callback (destroy)" editboxes). They can be implemented in a separate unit, and this unit would have to be included in the uses section of *DynTFTHandlers.pas* file, via the *DynTFTHandlersAdditionalUnits.inc* file. In most of the cases, either using an SD card or another tpye of storage, this storage medium has to be initialized and the files be already open for when the RTTI instruction execution engine would be run. If using both create and destroy instructions in the same UI (e.g. when creating/destroying components from manual creation groups), two files would have to be open at the same time. If only create instructions are used, a single file would suffice. Also, make sure to close the file after UI initialization, if not using manual creation groups.

Here is an example of two callbacks, using FAT32 library:

```
procedure RTTICreateCallback(ABinaryComponentsData: PDWordArray; ByteCount: TSInt);
begin
  case ByteCount of
    CRTTI_DataCallBackArg_ResetFilePointer :
      //Reset to the first instruction! (can be > 0 if using header)
      FAT32_Seek(RTTI_Create_Handle, 0);

    CRTTI_DataCallBackArg_GetDataBuildNumber :
      ABinaryComponentsData^[0] := C_DynTFTUI_BuildNumber

    else
      if FAT32_Read(RTTI_Create_Handle,
                    PByteArray(TPtr(ABinaryComponentsData)),
                    ByteCount) = -1 then
      begin
        //stops execution, if this is the first DWord of an instruction
        ABinaryComponentsData^[0] := 0;
        {$IFDEF IsDesktop}
          DynTFT_DebugConsole('Reached end of file, or file not found in RTTI create.');
        {$ELSE}
          //make sure FAT32 is initialized
          DynTFTDisplayErrorMessage(CRTTIDATAPROVIDERREADERR, CL_RED);
        {$ENDIF}
      end;
  end; //case
end;


procedure RTTIDestroyCallback(ABinaryComponentsData: PDWordArray; ByteCount: TSInt);
begin
  case ByteCount of
    CRTTI_DataCallBackArg_ResetFilePointer :
      //Reset to the first instruction! (can be > 0 if using header)
      FAT32_Seek(RTTI_Destroy_Handle, 0);

    CRTTI_DataCallBackArg_GetDataBuildNumber :
      ABinaryComponentsData^[0] := C_DynTFTUI_BuildNumber

    else
      if FAT32_Read(RTTI_Destroy_Handle,
                    PByteArray(TPtr(ABinaryComponentsData)),
                    ByteCount) = -1 then
      begin
        //stops execution, if this is the first DWord of an instruction
        ABinaryComponentsData^[0] := 0;
        {$IFDEF IsDesktop}
          DynTFT_DebugConsole('Reached end of file, or file not found in RTTI destroy.');
        {$ELSE}
          //make sure FAT32 is initialized
          DynTFTDisplayErrorMessage(CRTTIDATAPROVIDERREADERR, CL_RED);
        {$ENDIF}
      end;
  end; //case
end;
```

The values of `RTTI_Create_Handle` and `RTTI_Destroy_Handle` variables would be returned by calls to `FAT32_Open` function. The way these callbacks are called, depends on the data they provide. Typically, a call with `ByteCount` set to 0, would be made at the start of any batch of instructions. This is to reset the file pointer to the first instruction. The above examples assume that .dyntftui files contain no header, so the first instruction would be located at offset 0. For every create/set instruction, a call with `ByteCount` set to 4, would be made, to fetch a new instruction, then another call with `ByteCount` set to the length specified by the instruction, to get its content. In case of a read error, the returned value has to be set to 0, to stop the execution.

If the project uses Dany's FAT32 libary, the callbacks would look like these:

```
procedure RTTICreateCallback(ABinaryComponentsData: PDWordArray; ByteCount: TSInt);
{$IFDEF IsDesktop}
  var
    TempBuffer: TArr4096Byte;
{$ENDIF}
begin
  case ByteCount of
    CRTTI_DataCallBackArg_ResetFilePointer :
      //Reset to the first instruction! (can be > 0 if using header)
      FAT32_Seek(RTTI_Create_FileVar, 0);

    CRTTI_DataCallBackArg_GetDataBuildNumber :
      ABinaryComponentsData^[0] := C_DynTFTUI_BuildNumber

    else
    begin
      {$IFDEF IsDesktop}
        if Fat32_ReadBuffer(RTTI_Create_FileVar,
                            TempBuffer,
                            ByteCount) <= 0 then
      {$ELSE}
        if Fat32_ReadBuffer(RTTI_Create_FileVar,
                            PByte(TPtr(ABinaryComponentsData)),
                            ByteCount) <= 0 then
      {$ENDIF}
      begin
        //stops execution, if this is the first DWord of an instruction
        ABinaryComponentsData^[0] := 0;
        {$IFDEF IsDesktop}
          DynTFT_DebugConsole('Reached end of file, or file not found in RTTI create.');
        {$ELSE}
          //make sure FAT32 is initialized
          DynTFTDisplayErrorMessage(CRTTIDATAPROVIDERREADERR, CL_RED);
        {$ENDIF}
      end
      else
      {$IFDEF IsDesktop}
        memcpy(PByte(TPtr(ABinaryComponentsData)), @TempBuffer, ByteCount);
      {$ENDIF}
    end;
  end; //case
end;


procedure RTTIDestroyCallback(ABinaryComponentsData: PDWordArray; ByteCount: TSInt);
{$IFDEF IsDesktop}
  var
    TempBuffer: TArr4096Byte;
{$ENDIF}
begin
  case ByteCount of
    CRTTI_DataCallBackArg_ResetFilePointer :
      //Reset to the first instruction! (can be > 0 if using header)
      FAT32_Seek(RTTI_Destroy_FileVar, 0);

    CRTTI_DataCallBackArg_GetDataBuildNumber :
      ABinaryComponentsData^[0] := C_DynTFTUI_BuildNumber

    else
    begin
      {$IFDEF IsDesktop}
        if Fat32_ReadBuffer(RTTI_Destroy_FileVar,
                            TempBuffer,
                            ByteCount) <= 0 then
      {$ELSE}
```

```
        if Fat32_ReadBuffer(RTTI_Destroy_FileVar,
                            PByte(TPtr(ABinaryComponentsData)),
                            ByteCount) <= 0 then
      {$ENDIF}
      begin
        //stops execution, if this is the first DWord of an instruction
        ABinaryComponentsData^[0] := 0;
        {$IFDEF IsDesktop}
          DynTFT_DebugConsole('Reached end of file, or file not found in RTTI destroy.');
        {$ELSE}
          //make sure FAT32 is initialized
          DynTFTDisplayErrorMessage(CRTTIDATAPROVIDERREADERR, CL_RED);
        {$ENDIF}
      end
      else
      {$IFDEF IsDesktop}
        memcpy(PByte(TPtr(ABinaryComponentsData)), @TempBuffer, ByteCount);
      {$ENDIF}
    end;
  end; //case
end;
```

Few tips when using RTTI and binary components:

- RTTI and binary components work by overwriting memory at predefined locations.
- In case of reading handler addresses from .lst files, always make sure they are in sync with the generated code (and .dyntftui files). Otherwise, undesired memory locations will be overwritten.
If other memory locations are overwritten, input pins may switch to output pins or output pins may switch state, or whatever register may be overwritten etc., causing unwanted effects.
- If using dyntftui files, make sure they are updated to SD card / USB Drive, before starting the board (at least before using them in GUI).
- Use DynTFTCodeGen to see if generated files are in sync with .lst files. Read DynTFTCodeGen's documentation (this file), to understand how it works. Always rebuild the Deksktop/MCU project(s) after generating files with DynTFTCodeGen.
- The .lst monitoring feature of DynTFTCodeGen is not perfect and it may not always report if .lst files are not in sync with the generated code.
- Users may also forget to rebuild a project after regenerating output files. In that case, DynTFTCodeGen reports that .lst files are in sync, but it doesn't know that the binary (exe/hex) was not rebuilt.
- Beware that if regenerating output files (DynTFTGUIObjects.pas, DynTFTHandlers.pas, *.dyntftui), when configured to read handler addresses from .lst files, the current state of .lst files is used.
- Always rebuild, not simply compile projects.
- Make sure RTTI projects have the "RTTIREG" compiler directive, defined at project level.
- Make sure RTTI projects have the profile name, set as compiler directive, defined at project level.
- Use *DynTFTHandlersAdditionalUnits.inc* file to add font definitions to projects (see chapter 11).
- Different combinations of "HasVarInGUIObjects" and "CreatedAtStartup" properties for multiple components, will have different effects on RAM and Flash usage.
- When creating a new Desktop/MCU project, which will be configured to use handler addresses from .lst files, DynTFTCodeGen won't be able to generate output files if directly configured to use the .lst file. This is because, the .lst has to exist so that DynTFTCodeGen can read it. In this case, please set that particular profile, to use array of addresses, which will allow a first build of the project, which will result in creating the .lst file. After that, the profile can be set to use .lst files.
- As a safety measure, to prevent out of date instructions to execute, DynTFTCodeGen adds a build number to every batch of instructions (code or .dyntftui file), which verifies if it matches a generated code constant. In case of a mismatch, instruction execution stops with an error message.

11. Additional units in DynTFT projects

In addition to generating *DynTFTGUI.pas*, *DynTFTGUIObjects.pas*, *DynTFTonts.pas* and *DynTFTHandlers.pas* from scratch, DynTFTCodeGen can edit an existing *DynTFTHandlers.pas* up to a certain extent, making sure user code is preserved. It expects that user code would be placed in *DynTFTHandlers.pas* , only inside generated event handlers (this includes adding local variables). The other parts of the code from *DynTFTHandlers.pas* , are almost completely regenerated at every editing operation by DynTFTCodeGen. When users have to provide additional code, which has to be used from implemented event handlers, the available options are to include new units in the uses section of *DynTFTHandlers.pas* , using "DynTFTHandlersAdditionalUnits.inc" or including the code via another include directive/file, "DynTFTHandlersAdditionalCode.inc". DynTFTCodeGen always expects that, if using *DynTFTHandlersAdditionalCode.inc*, it has to be defined as `{$I DynTFTHandlersAdditionalCode.inc}` , two lines after the line containing the "implementation" keyword from *DynTFTHandlers.pas* file, with no other content in between and no indentation:

```
//CodegenSym:AllBinHandlersEnd

implementation

{$I DynTFTHandlersAdditionalCode.inc}

//CodegenSym:UpdateBinHandlersProcBegin
```

When placing procedures and functions inside *DynTFTHandlersAdditionalCode.inc* file, which have to be visibile from other units, their headers will have to be added to *DynTFTHandlersAdditionalCodeInInterface.inc* and the "UseHandlersAdditionalCodeInInterface" compiler directive be defined at project level.

In the case of using different TFT libraries, rather than the provided mikro one, they can be added to the uses section, with their names defined in *UserDrawingUnits.inc* file. To include this file, the "UserTFTCommands" compiler directive has to be defined at project level.

```
  {$IFNDEF UserTFTCommands}
    {$IFDEF IsDesktop} , TFT {$ENDIF}
  {$ELSE}
    , {$I UserDrawingUnits.inc}
  {$ENDIF}
  {$I DynTFTHandlersAdditionalUnits.inc} ;

  {$IFDEF UseHandlersAdditionalCodeInInterface}
    {$I DynTFTHandlersAdditionalCodeInInterface.inc} //external functions, vars, consts
  {$ENDIF}
```

The *DynTFTHandlersAdditionalUnits.inc* file can also be used to include files containing "external resources", required by a DynTFT project, e.g. font definitions. For example, if a font is defined in a file, called *ExternalResources.mpas* , it can be included like this:

```
{$IFNDEF IsMCU}
  , ExternalResources
{$ENDIF}
```

It has to be included for MCU only, because on desktop simulators, fonts are already defined in *DynTFTonts.pas* file.

In the current version, DynTFTCodeGen does not update the beginning section of *DynTFTHandlers.pas* file, which contains all these directives and include statements. Because of this, the beginning section of existing *DynTFTHandlers.pas* files would have to be manually updated to the latest version from *DynTFTHandlers_Beginning.txt* file from DynTFTCodeGen's *OutputParts* directory (mentioned in chapter 2).