# SA4041 Developer's Guide

## Introduction

This guide is an overview of the installed software and Solantro's programming methods. For more information please refer to:

- Software Development Environment (SDE) Installation Guide
- Solantro Bootloader Manual
- Helios Test and Control User's Guide
- Git version control documentation
- Help – Eclipse IDE: Workbench User Guide

It would be beneficial to the reader to have Eclipse and Helios installed along with a project with source code from Solantro to explore the material referenced in this overview. For help with Eclipse: open the program and go to Help then Welcome Contents where different tutorials and configuration settings can be found.

### About the installed software**:**

- Eclipse is the Integrated Development Environment (IDE) used to edit and compile the C code for the RISC core in Solantro's processors. The installer for the Solantro Edition installs: Cygwin, WinPcap, Java Runtime Environment and Acrobat Reader if they are not already present.
- Helios is a stand-alone Windows GUI application which uses a COM port to communicate with the firmware running on the Solantro processor to observe and change, software variables and hardware registers. It is also used to program the processor.
- Python 3.7 is used by build scripts to create the XML description files and place them in the build projects directory. The xmltodict package needs to be added to Python using pip for the build scripts to work.
- USB to UART TTL drivers are needed on most computers. For power supply applications, isolated converters should be used while the development boards may use a direct USB to UART bridge IC.
- PuTTY is a terminal emulator which uses a COM port to communicate with the Solantro controllers. The processor has a small ROM bootloader and monitor program to help get things started. The UART is typically used to display debug messages when the firmware is loaded and running.
- Git is a version control system. It is optional but highly recommended since it helps to track changes made to files and folders. The build scripts will additionally increment the version number and add a version tag when changes are made, and commits are performed.

### Recommended Applications:

- **Sourcetree** is a free Git client popular at Solantro along with using Bitbucket as a free repository hosting service.
- **TortoiseGit** is a Windows shell interface so it adds Git commands directly to Windows Explorer and provides overlay icons which indicate the status of the files and folders.
- **Sublime Text** and **Notepad++** are sophisticated text editors which are much better than Notepad or Wordpad for general use and highly recommended.

## Software Flow and Hardware Connections:

Figure 1 shows the software flow and hardware connections for development and testing. It starts with a C project folder which is compiled using the Eclipse IDE. An XML file is created by building the project in Eclipse. Each project and its firmware versions have the variable and register descriptions in a compressed XML file stored in one of two main folders: Helios Build Projects or Helios Release Projects. Helios will let you browse to set these folders, so the application and folders do not have to be in a specific location. Two COM ports are used for communication by Helios and PuTTY for the purpose of development, test and control of the hardware platform.

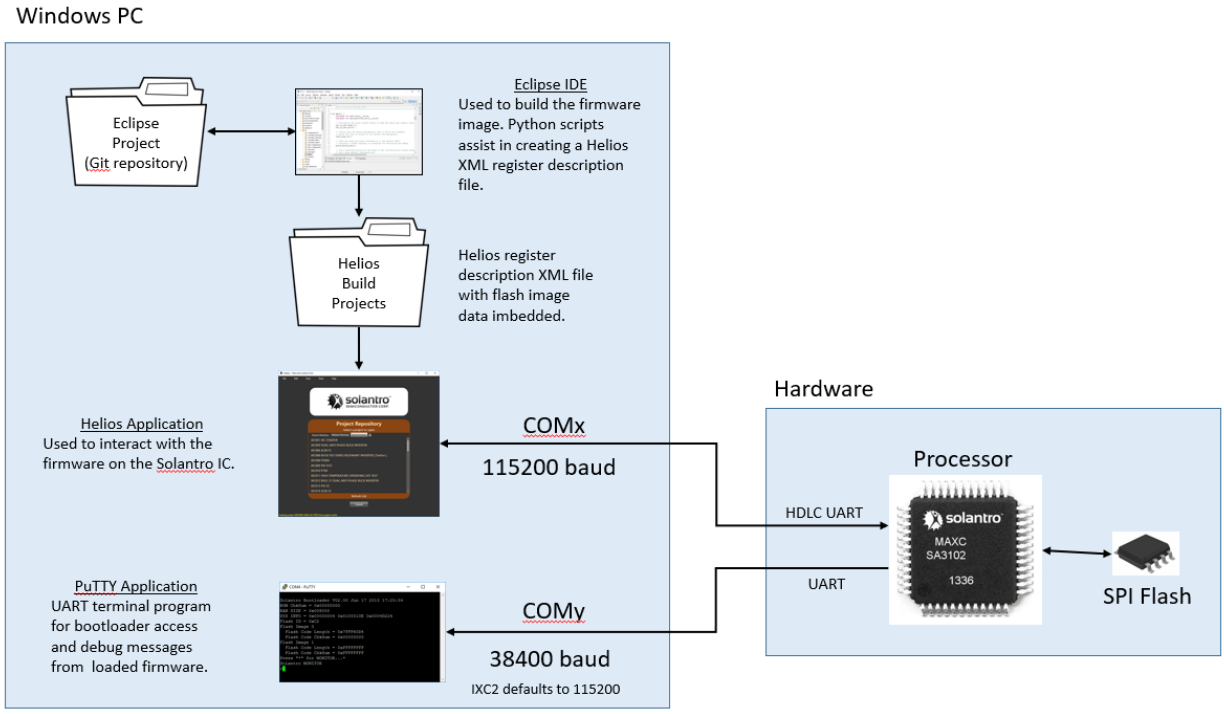
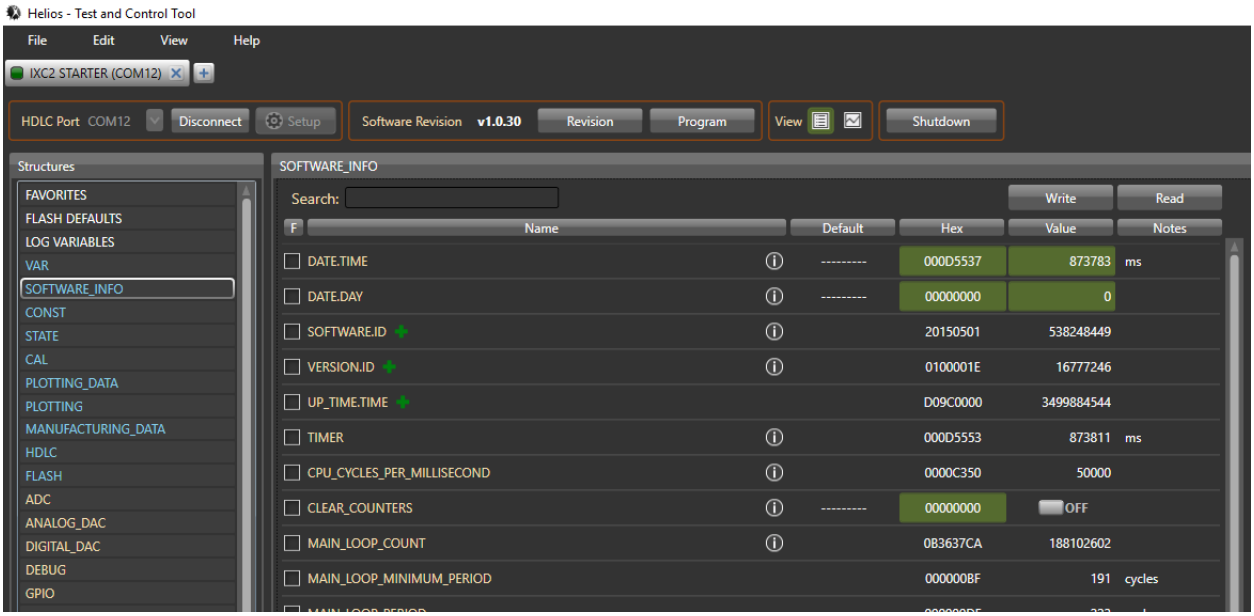



**Figure 1- Software flow and Hardware connections**

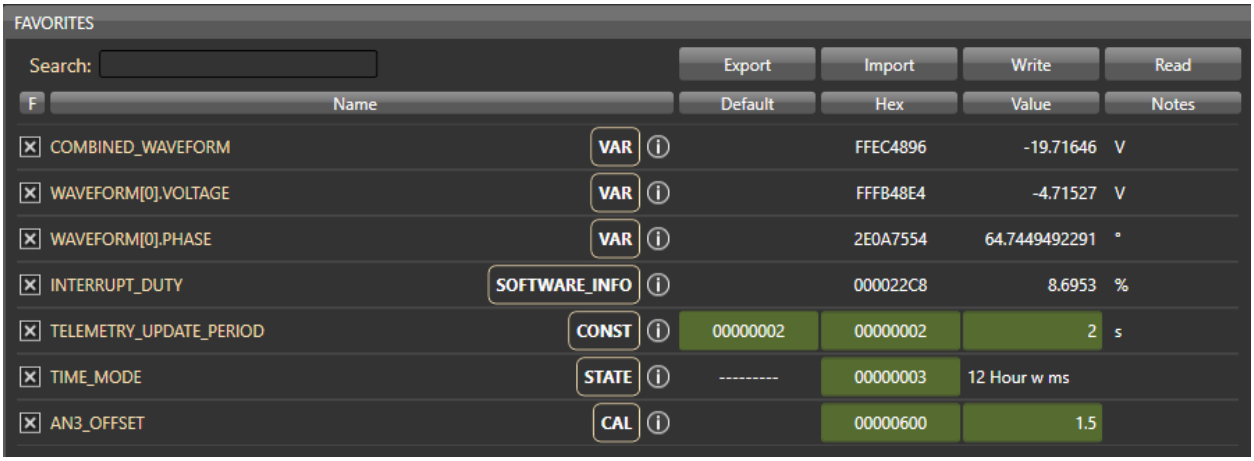
# Helios

The GUI interface is used to read and write software variables and hardware registers among other things such as plotting and programming. More information is available in the user guide, but a cropped screen capture and brief tour is presented here for reference.




The left Structures pane lists the firmware global variable structures in blue and the hardware peripheral structures in yellow. The SOFTWARE_INFO structure is selected which brings up a view of the SOFTWARE_INFO variables on the right. The name column may have an information or help icon on the right side which pops up a window when the mouse hovers over it for additional information. The Default column allows the user to save the value into the flash to be read when the hardware reboots. This can be used to change the default values without re-compiling. Values which are read only have the default section disabled. The Hex column represents the integer values stored in the hardware and the Value column is converted based upon any unit multipliers or the radix position indicated in the source code. Values which can be changed by the user have a green background. The Notes column serves as display area for the units.

## Favorites



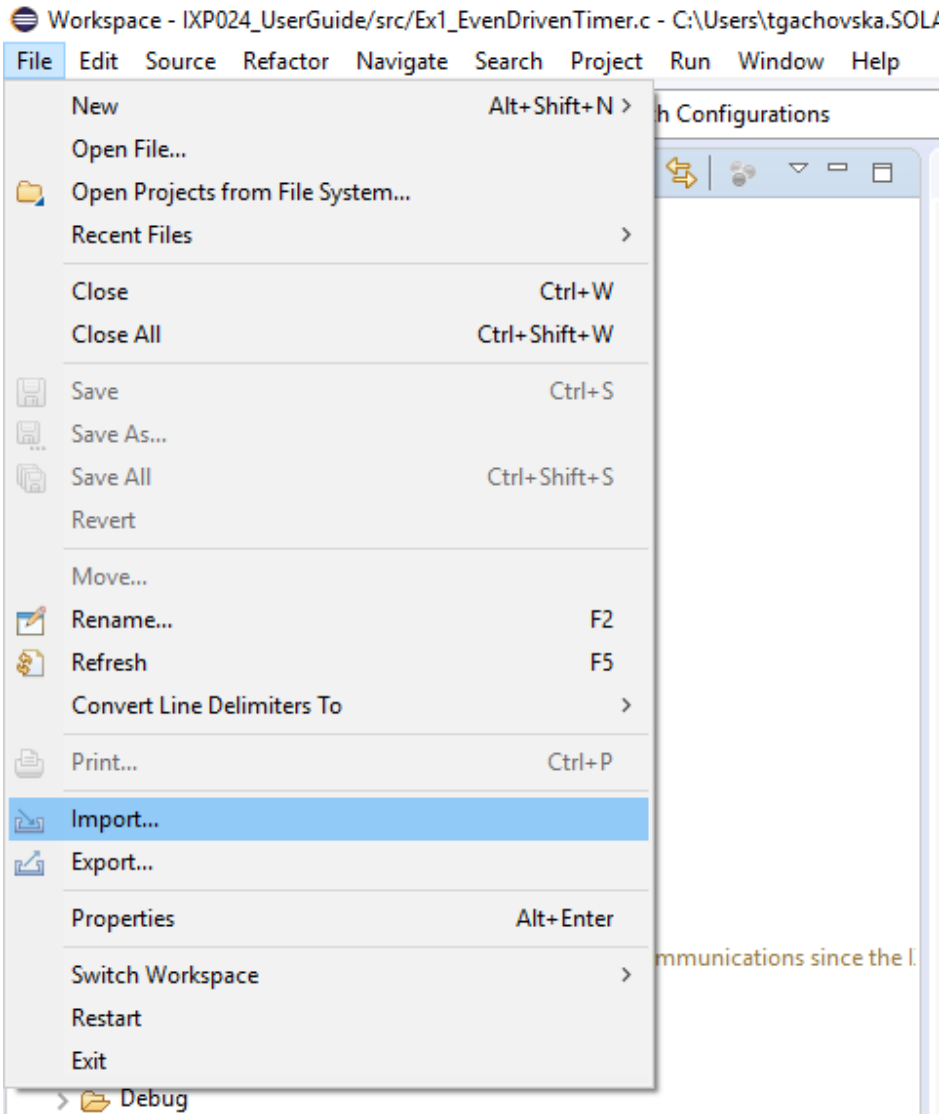

The favorites pane allows you to select variables from different structures to be displayed together. The picture above also serves as an example of the different properties given to various variables.

The VAR and SOFTWARE_INFO structures contain examples of read only variables with units of Volts, Degrees and Percentage. While the variables are stored as 32-bit integers as seen in the Hex column, they are displayed as real values in the value column. The CONST structure variable is an example of a read / write variable with the flash default set (user non-volatile default). The STATE variable is an example of an enumerated variable which has a value of 3 but it is displayed as text in the Value column. The CAL structure variable is another example of a read / write variable, but it is not possible to set a flash default in this case.

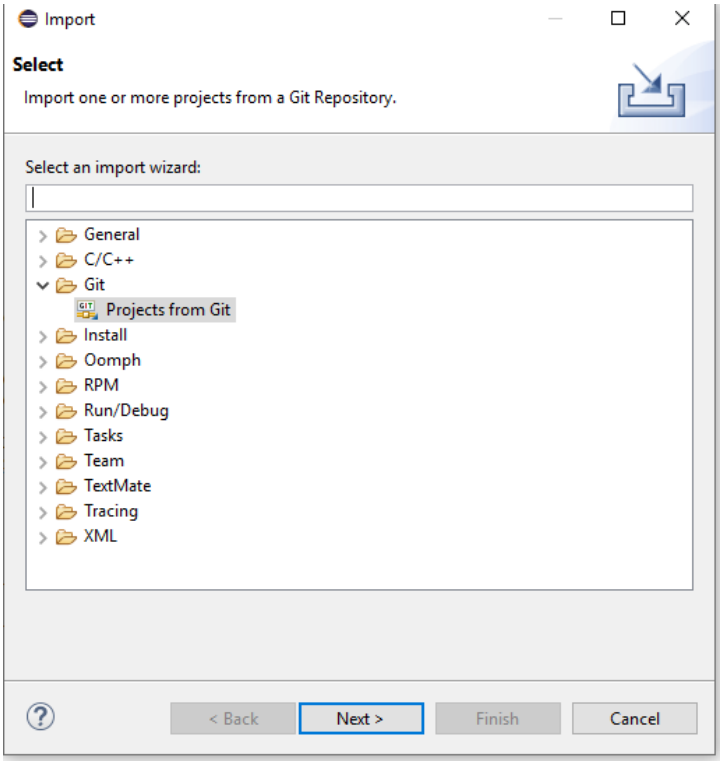
More information is available in the Helios Users Guide and later in this guide in the section variable naming conventions.

# Getting Started

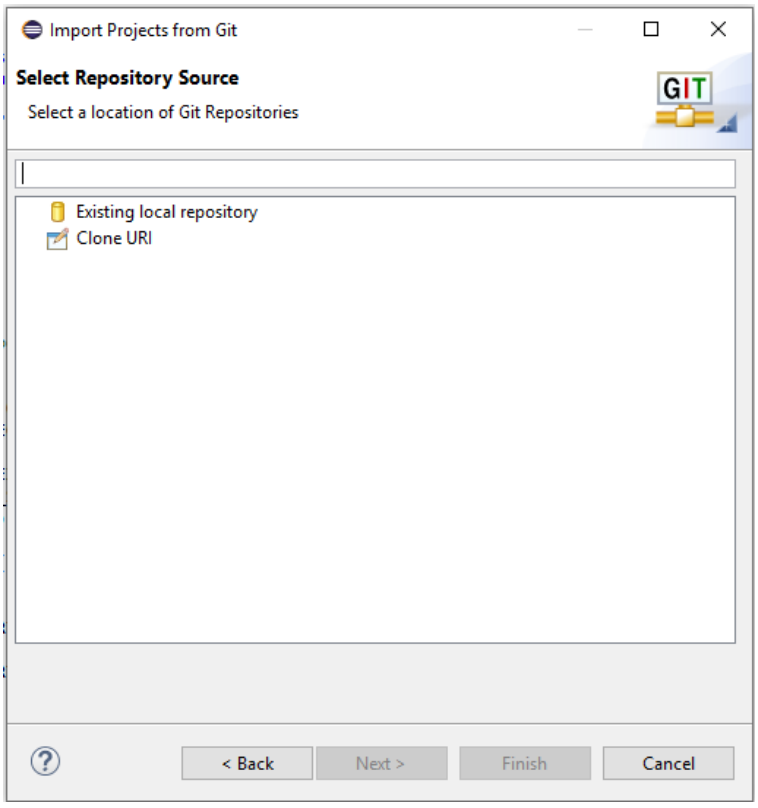
Download one of Solantro's projects to import it into Eclipse. The import can be done by going "File" then "Import…" as shown in figure below:




Then select "Projects from Git" as shown and press "Next".

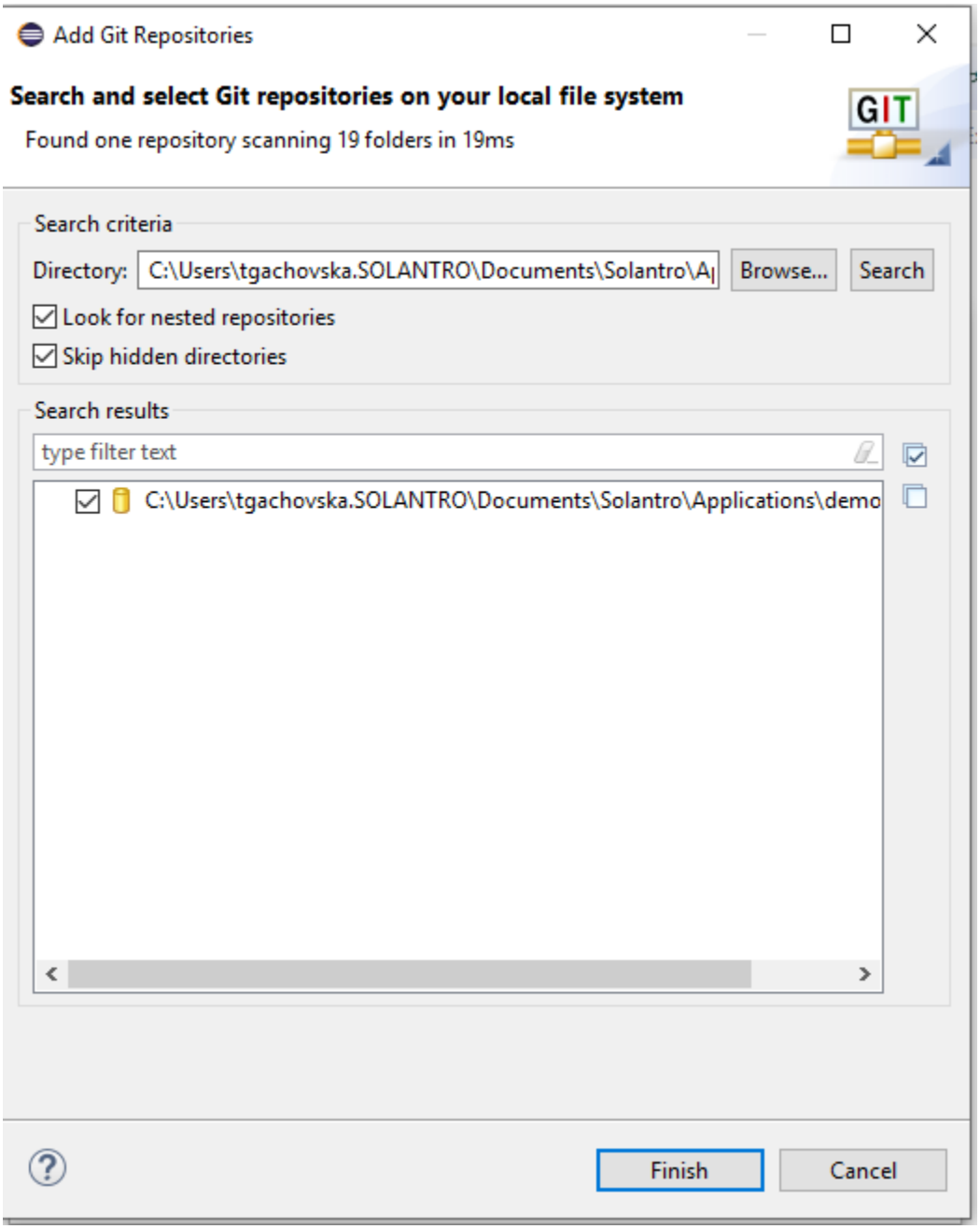


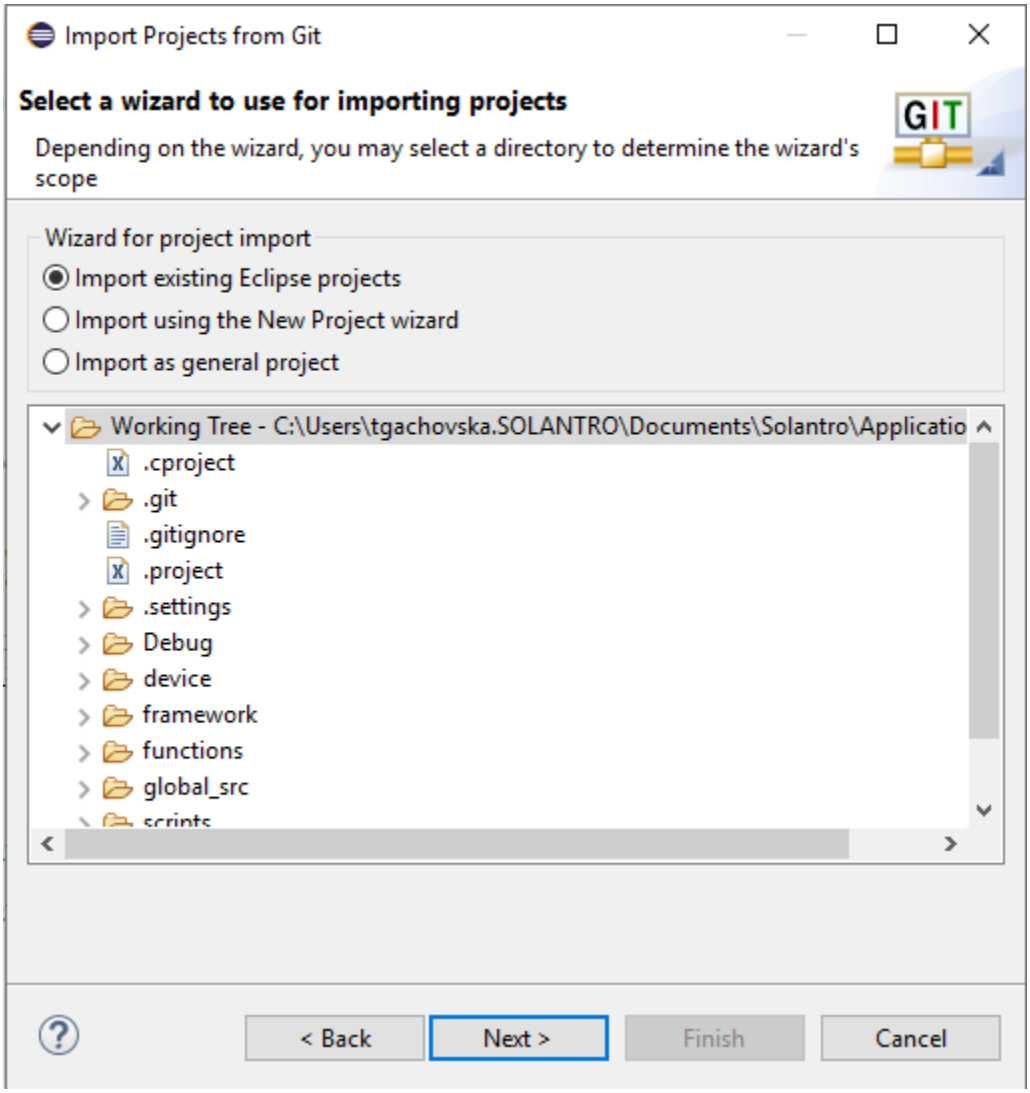

Select “Existing local repository”.

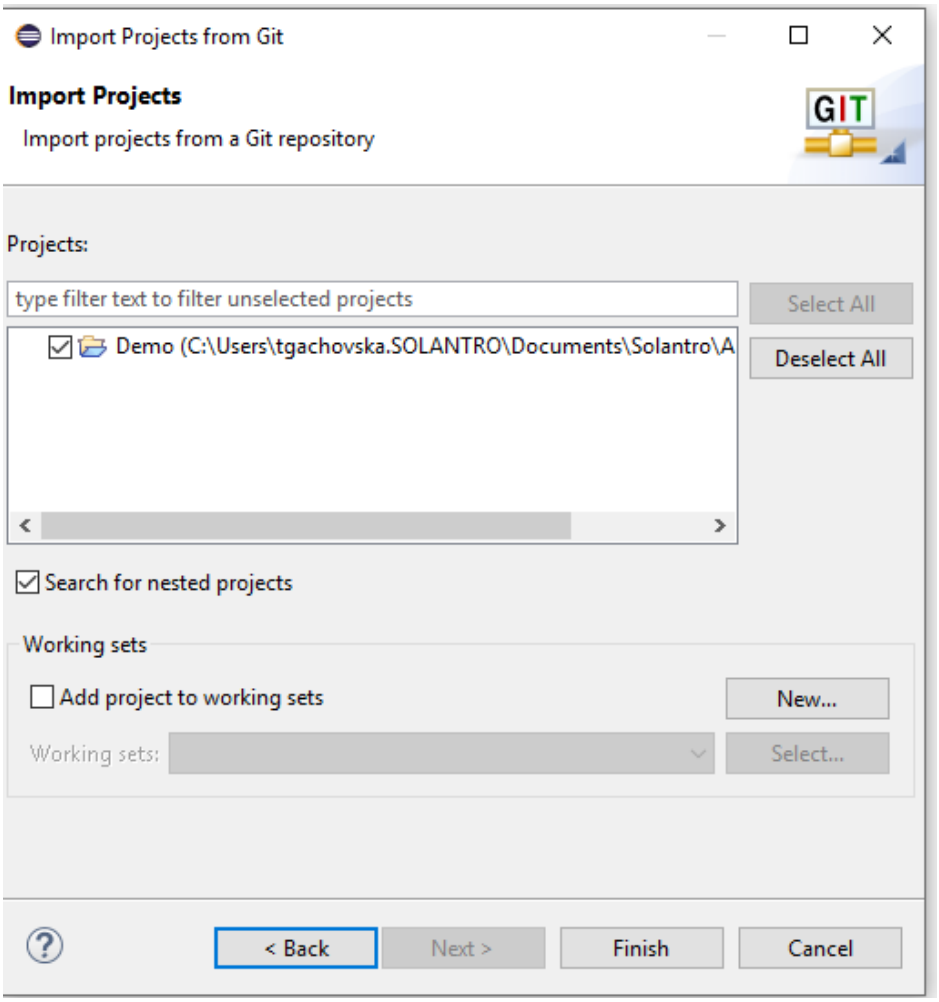



Then you should select the repository Source by “Add…” the path.

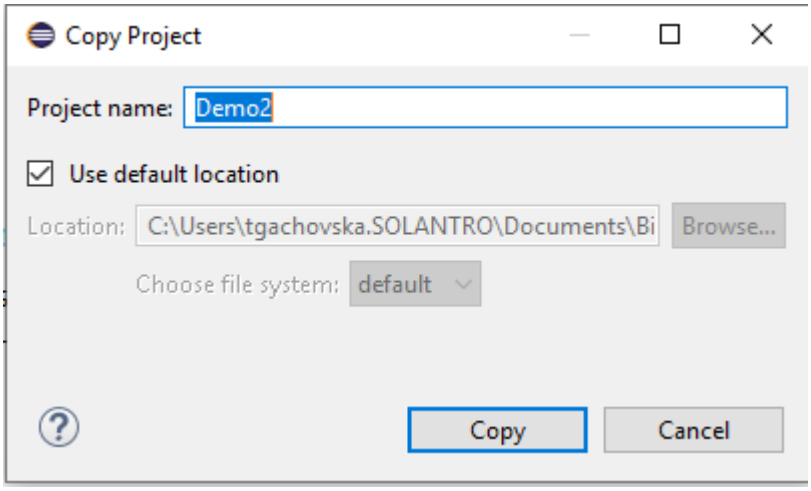

Select the path where Demo project is selected.


Add Git Repositories
Search and select Git repositories on your local file system
Found one repository scanning 19 folders in 19ms
GIT
Search criteria
Directory: C:\Users\tgachovska.SOLANTRO\Documents\Solantro\A
Browse...
Search
Look for nested repositories
Skip hidden directories
Search results
type filter text
C:\Users\tgachovska.SOLANTRO\Documents\Solantro\Applications\demo
Finish
Cancel


Import Projects from Git
Select a wizard to use for importing projects
Depending on the wizard, you may select a directory to determine the wizard's scope
GIT
Wizard for project import
Import existing Eclipse projects
Import using the New Project wizard
Import as general project
Working Tree - C:\Users\tgachovska.SOLANTRO\Documents\Solantro\Applicatio
.cproject
.git
.gitignore
.project
.settings
Debug
device
framework
functions
global_src
< Back
Next >
Finish
Cancel

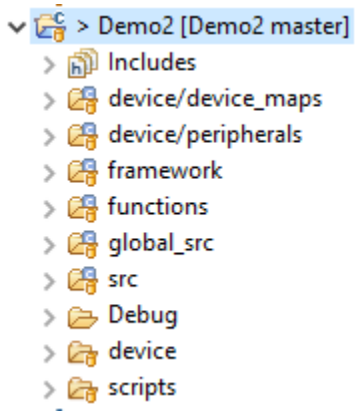



This Demo project can be used as a new project by duplicating it in Eclipse. In the Eclipse Project Explorer, right click on the project to copy and then right click to paste. Hence, you will be met with the following dialog box. Select a new project name and location.



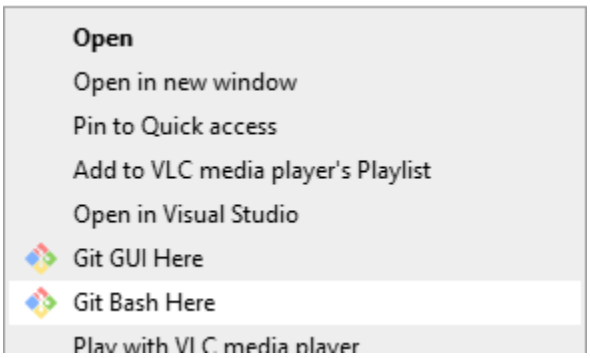

The default location will be the workspace, but you may have your projects organized in other folders so you can unclick the default location and browse to a new empty folder in your filesystem.

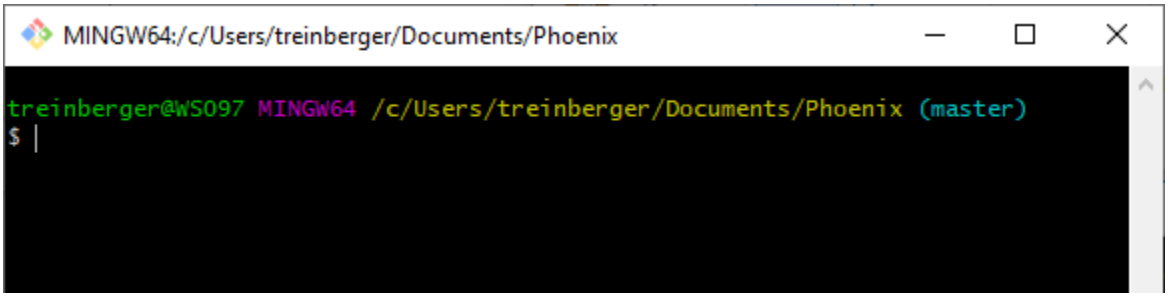
Alternately, you can duplicate a project folder in the file system to make a new project but it will not import into the same workspace due to having the same original name in the .project file. However, you can then edit this file to change the projectDescription name property. You will now be able to import it as a new project. The Demo 2 project will include the duplicated folders:




Edit the version.h file (“src” folder) to update the SOFTWARE_ID_VALUE (usually start date YYYYMMDD in hex), PROJECT_NAME and PROJECT_ALIAS. You may also want to reset the version numbers.

Git repositories may be synchronized with a remote server, sometimes called the origin. When a project is copied, it will still point to the same remote if it has one. To avoid pushing the changes from your new project to the remote repository of the original project, you should delete the remote with the following command using either the command window or a git bash shell. If Git was installed from the Solantro guide, then the git bash shell will be available with a right click on the project folder in Windows Explorer.




```
git remote rm origin
```

You can add a new origin in the bash shell or later using Sourcetree:

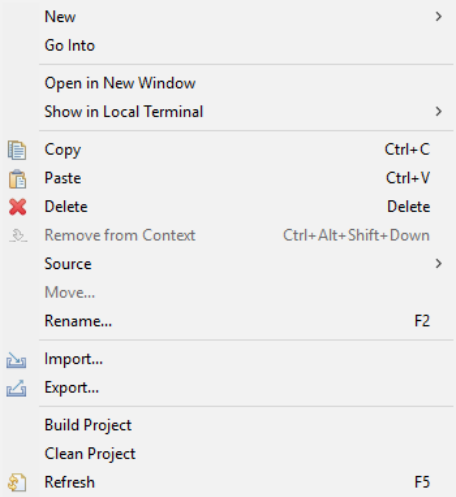
```
git remote add origin <url>
```

The history of any previous commits can be removed with the following commands in the git bash shell.

```
git tag | xargs git tag -d    # Removes all tags
git branch -m temp            # Give the current branch a temp name
git checkout --orphan master  # Use the current files as the new master branch
git add .                     # Add in case any have been added or modified
git commit -m "Using project x version y as the starting point."
git branch -D temp            # Delete other branches to clean up
git gc --aggressive --prune=now  # Remove non referenced commits
```

# Build process: Test the copied project

The development process in Eclipse relies on having Python 3 installed and Helios needs to have been run once to designate the location of the Build Projects Folder. It all begins by selecting “Build Project” from the menu or toolbar. NOTE: “Clean Project” is recommended before the build since we have encountered situations when the source code was changed but the object file was not updated. You can access these options, “Clean Project” and “Build Project”, by right-clicking on the name of the project in Project Explorer.




The binary is compiled and linked which is then followed by a post-build script which creates an XML register description file for Helios and creates other files in the Debug folder which may come in handy. The script and build messages are displayed in the Console window as shown.

```
   Text    data     bss     dec     hex filename
  10856    5540     172   16568    40b8 RAM_image.exe
XML file being saved to: C:\Users\user\Documents\IXC2 DEMO\master
Version: v3.1.6 created.
Postbuild elapsed time: 1.685s
18:48:41 Build Finished. 0 errors, 0 warnings. (took 22s.197ms)
```

Refer to the Helios User Guide or Bootloader User Manual to learn how to program the devices.

# Project Tour:

An Eclipse project is imported through the Project Explorer pane. It may or may not be copied into the file system workspace folder. The Helios Demo project is used here as an example. Since the project is under Git revision control, the name of the repository and the branch is displayed after the name.

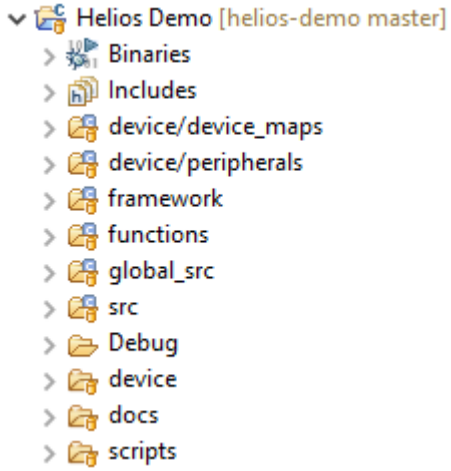



**Figure 2- An example of project folders**

The **Binaries** and **Includes** folders are virtual folders which indicate the files referenced to build the binary and the files referenced through the include paths.

The folders involved in the build have a C in the upper right-hand corner of the icon. The first four of these folders are typically provided by Solantro while **global_src** and **src** folders are typically left for the developer to add to or modify. The descriptions follow:

- **device/device_maps** contains defines and functions related to the processor such as the memory map for the peripherals and interrupt mask bits as well as the supported chip IDs. As indicated by the name, the folder in the filesystem is inside the device folder.
- **device/peripherals** contain the header files for the peripheral registers. Each file contains the typedef structure for a specific peripheral on the AMBA bus. As indicated by the name, the folder in the filesystem is inside the device folder.
- **framework** contains a group of files which typically use the peripherals to carry out their functions. Considered application specific functions they include the functions to print debug messages, access the flash, and exchange data with Helios etc.
- **functions** contain more general purpose defines and functions which may be useful to the developer. Solantro's code makes heavy use of the defines in solantro_constants.h.
- **global_src** contains structures for the global variables which will appear in Helios. This group is a convention only. The file names or the fact that they are in the global_src folder has no impact on their actual functionality. Most projects contain files which group the variables into 4 categories. Other projects may contain more files or categories to keep the number of variables in each to a manageable number. See the previous Helios section which displays a few structure variables.
    - ***calibration_data*** contains variables which are copied into the flash to be stored as non-volatile values. These values are usually different from board to board and initialized from the flash when the firmware boots.
    - ***constants*** are read/write variables typically used to configure limits or any values which might be common to all identical boards. Examples are voltage thresholds or timing constants.
    - ***states*** are variables which may represent a single bit or an enumeration. These may be set by the user (read / write) or determined by the firmware in which case they should be flagged as (read only). Helios presents single bit variables as an on/off button and enumerated variables as a drop-down list. A read only enumerated list will display the text value in Helios.
    - ***variables*** are usually calculated by the software and should be flagged as read only. The values are usually captured by the plotting routines, so they can be graphed in Helios. Examples are measured voltages, calculated power or other control variables.
- **src** folder contains the other files and functions created by the developer to control their design including the main and interrupt handler functions.

Additional files and folders can be created to help organize the code. The file names or organization does not dictate how they are used by the compiler nor the build scripts when they generate an output for Helios.

Other folders in the picture include:

- **Debug** folder contains the output from the build process which includes the object files, disassembly file, bootloader programming text and the binary executable files.
- **device** folder is the parent folder to the device_maps and peripherals but it also contains the linker scripts and, in some cases, some device documentation.

- **docs** folder may contain the project documents or an Excel file to generate a programming string for the manufacturing information.
- **scripts** folder contains the Python 3 scripts used in the build process in addition to hook scripts used for the commit steps of a Git version-controlled repository.

## Helios Communication

Helios communicates with the firmware through the HDLC UART peripheral. The hdlc_poll() function in the main loop is used to decode the address and data so that information can be passed between the Helios application and the flash, RAM or AMBA peripherals in the IC. The address is normally hidden in Helios, but the field visibility can be turned on as seen in the picture below. The Hex column represents the data portion of the packet.


The address decoding for the AMBA peripheral registers are the same as the physical memory map. The flash base address is configurable through a software define which is now set at 0x01000000. The global variables in the structures have an address which depends upon their order in the HDLC_FUNCTIONS_TYPE structure which is defined in the hdlc_configuration.h file.

More details on how the firmware is configured to work with Helios is covered in Appendix A

# Adding Variables to Helios

Variables can be added or deleted from the existing structures in the code without changing any other defines or hdlc_configuration files. The following section will explain in more detail the naming conventions and special comments.

## Variable naming conventions

The naming convention described here is not a requirement, however, it will allow you to get the most out of the Helios GUI. You may find that it also makes any math expressions in your code more readable and easier for other people to review or modify.

The variable definitions in the global structures take on the following form:

```
/// Some comment describing the variable for the user.
type the_name__units_precision; ///< More comments and attributes.
```

You can look at the code in Solantro's projects to see more examples in addition to the ones below without comments:

```
int BATTERY_VOLTAGE__V_U4F10;
int AC_FREQUENCY__Hz_U30E23;
int PV_POWER__W_S24E10;
```

The breakdown and description for each part is listed below:

- ***Type*** may be a signed or unsigned integer, struct or enumeration. If the type is an enumerated type, then Helios will present it as a drop-down list or text.
- ***C Identifier*** consists of the name and two optional parts:
    - **Name**: text may include single underscores. A double underscore separates the name from the optional parts.
    - **Units**: cannot contain underscores but special text may be displayed as symbols. See the units section for more information.
    - **Precision**: starts with U or S and indicates the number of integer and fractional bits used to store the value and indicate the radix point. See the precision section for more information.
- ***Special comments*** use the Doxygen commenting format with the most used being /// and ///<. These comments do not affect the compiled code, but they will pass description information and properties on to Helios.

An example below as it is found in some code. This variable represents the voltage on pin AN3 after it is scaled by the ADC reading in another part of the firmware.

```
/// Scaled voltage based upon calibration values. The range should be
/// 0 V to 1.8 V but we make it signed to handle offsets.
int AN3__V_S3F12; ///< -RO
```

Figure 3 shows a Helios view of the AN3 variable along with the Register Description pop-up which comes up when the mouse hovers over the (i) icon.

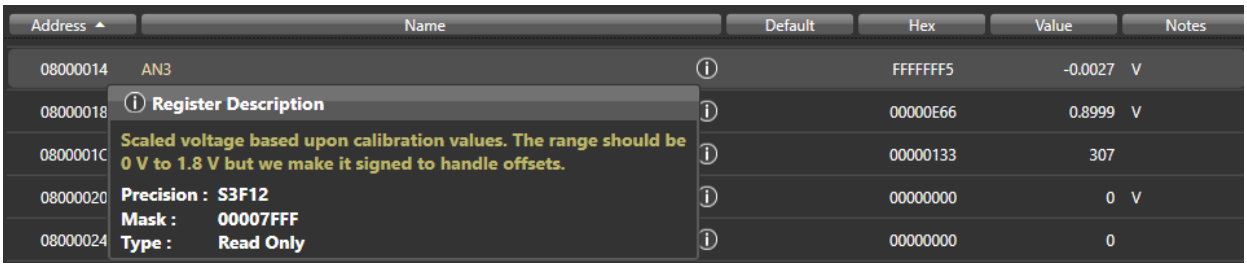



**Figure 3- Helios example for AN3 variable**

As you can see, the comments which precede the variable definition are used as information or help comments in Helios. AN3 shows up in the name column and the units V shows up in the Notes column on the right. S3F12 indicates the precision that should be used by Helios. For this example: signed with three integer bits (including the sign bit) and 12 fractional bits. This lets Helios translate the data bits into a real value. The trailing comment of -RO is concatenated to the other comment but since it is one of the specially defined comments, it is not displayed. It instead tells Helios that the variable should be read only and not let the user change the value in the GUI.

***Note***: the precision text does not have any effect on how it is stored in the processor which is an int (32-bit signed value).

### Unit Symbols

The unit's text in the variable definition can be preceded by a number which acts as a multiplier and the unit text can define special characters which are not legal as a C code identifier but will display nicely in the Helios application. The following unit text will be displayed as a symbol.

| Text | Symbol |
|---|---|
| PERCENT | % |
| DEG | ° |
| DIV | / |
| SUM | ∑ |
| SQRT | √ |
| MICRO | µ |
| OHM | Ω |

See the favorites screenshot in the previous Helios section for examples of the following variables.

```
VAR.COMBINED_WAVEFORM__V_S32E16
VAR.WAVEFORM[0].VOLTAGE__V_S27E16
VAR.WAVEFORM[0].PHASE__360DEG_U32E32
SOFTWARE_INFO.INTERRUPT_DUTY__PERCENT_U17E10
```

**Precision**

The precision text has two formats:

**E** format starts with S or U and then the total number of bits (including the sign bit and fractional bits) which is used to store the value followed by E and then the number of fractional bits. S15E12 for example represents a signed value using 15 bits. 12 bits are for the fractional part, 1 bit is for the sign which leaves 2 bits for the integer part.

**F** format also starts with S or U and then the number of integer bits (including the sign bit) followed by F and then the number of fractional bits. S3F12 for example also represents 12 bits for the fractional part, 1 bit for the sign which leaves 2 bits for the integer part.

The F notation is easier for the developer to infer magnitude while the E notation is more useful in a mathematical division expression. The E notation can be used to imply leading or trailing zeros so larger or smaller numbers in fewer bits than F notation.

Since S15E12 is the same as S3F12 and using 15 bits to represent a number we can look at how it is translated into a real value. To start, a 15-bit unsigned number which ranges from 0 to 32676. As a signed number it ranges from -16384 to 16383. These are how they are stored in the processor. If 12 of the bits are fractional bits then the scaling value is $2^{12}$ or 4096. The real value range now becomes -4 to 3.99975 which is what would be displayed in Helios in the value column.

The E notation can be used for very large or small numbers because it explicitly indicates the number of significant bits and can imply leading or trailing zeros with the radix position. For example, U8E16 uses 8 bits but indicates the radix point is at bit 16 (moved 16 to the left). The only way to represent that is with 8 binary zeroes inserted between the radix point and the most significant bit. The full binary representation of the 8 bits would be 0.00000000nnnnnnnn. If x is the number representing the 8 significant bits, then the real value is $x / 2^{16}$. Trailing zeros can be represented with a negative radix which can be indicated with an underscore before the radix number. U8E_8 is again 8 significant bits but the radix point is at bit -8 (moves 8 to the right). The only way to represent that is with 8 binary zeroes inserted between the radix point and the least significant bit. The full binary representation of the 8 bits would be nnnnnnnnnn00000000. If x is the number representing the 8 significant bits, then the real value is $x / 2^{-8}$ or $x \cdot 2^{8}$.

In addition to letting Helios display real values using integer storage, the units and precision naming convention makes it easier to verify any formulas in the code and makes it easier to determine when 32-bit integer math may overflow and require a 64-bit operation with a (long long) typecast.

**Special comments**

The special comments include special flags starting with a dash. Their purpose is to give Helios more information about the variable. If a variable is read only then Helios won't let the user try to change it. If the variable should not be saved to the flash as a user default then the programmer can do that with the -NFD comment. Helios also displays variables based upon the user level. By assigning a user level to a variable, the programmer can select some to be part of a simpler interface where Helios will hide variables with a lower number. This table is not the complete list since some reserved comments have yet to be implemented in Helios and others are already defined in the code where they are needed. The developer relevant ones are listed here.

| Text | Meaning |
|---|---|
| -RO, -WO, -RW | -RW is the default which is read/write. -RO will tell Helios to not let the user change the value in the GUI. |
| -NFD | This tells Helios that the value should not be set/saved as a flash default. |
| -SHUTDOWN | Tells Helios that it can write a 1 to this variable to take some action. If this attribute is defined for a variable, then Helios will display a shutdown button next to the view icons for quick and easy access. Think of it as an emergency stop. It is up to the user to determine the action taken when the variable is set. |
| -UL(#) | Defines the user level in Helios. It gives the developer the ability to hide certain variables. Valid numbers are 1 to 5.<br>1 – (default) the lowest level. These will be hidden when the Helios user level is a higher number.<br>3 – The same visibility as the peripheral registers since they default to 3.<br>4 – Values which will still be visible when the peripheral registers are hidden. |

# Adding new structures

You can add your own structures quite easily by copying one of the files in global_src folder and renaming the structure definition for your own use. You will need to slightly modify the function names and other aspects of the functions but that should be straight forward. If not, then you can compare two files in the global_src folder to see what has been changed between them. Then add the equivalent parts to the hdlc_configuration files. Don't forget to add the #include for your new header file.

# More Structures and Functions

Projects may have additional structures to group variables and functions. The framework includes some optional ones which may be used such as MODBUS, CANBUS or APM control in addition to the common ones of HDLC_COMMUNICATION, MANUFACTURING_DATA and SOFTWARE_INFO.

The SOFTWARE_INFO stores the software ID and version number which Helios will read to determine which project and register description file to load when connecting. Functions in the code will also calculate some useful stats which are also stored in this structure. They include the percentage of time spent in the interrupt handler and how long it takes to go through the main loop. All of which can help to make sure you have the resources needed to execute your code properly.

The following functions are contained in the framework files and likely already called in the Solantro projects. The interested reader can look at the source code for more insight into their function and use.

- **calculate_main_loop_stats();** determines various values for SOFTWARE_INFO variables related to the main loop.
- **calculate_interrupt_stats**(); determines various values for SOFTWARE_INFO variables related to the interrupt handler.
- **update_uptime();** updates the date in millisecond which is also in the SOFTWARE_INFO variables and useful for delays or state machine timing. It may be in the main loop or interrupt loop but not both.
- **hdlc_poll();** handles Helios communications. It is called in the main loop.
- **debug_put_message("some string");** is used throughout the code, including the interrupt handler to place text in a buffer to be sent to the UART AKA PuTTY as debug or status messages using the debug_send_message function.
- **debug_send_message();** sends debug messages one character at a time. It is called in the main loop.
- **sample_the_plotting_data();** takes up to 4 pointers to capture the values in a memory buffer. Helios can set the pointers, capture triggers and read the data for plotting in the graphing window. The number of data points for storage is defined in configuration.h.

# Appendix A

This information is provided to help the developer understand the defines and functions present in the code which lets Helios and the Software with the help of the build scripts work together.

- **configuration.h** sets the flash memory map along with a few more constants used by other functions. In most cases, the values cannot or should not be changed.

```
#define NUMBER_OF_PLOTTING_SAMPLES  256

#define FLASH_SIZE_VALUE              0x40000   // 0x40000 = 256KB or 2Mb)
#define FLASH_HDLC_BASE_ADDRESS    0x01000000   // Was previously 0x80000000
#define FLASH_WRITE_ENABLE_KEY     0x0000FAAA   // Enables HDLC flash writes

// Flash memory map. Addresses are in bytes.
#define FLASH_IMAGE_0_BASE_ADDRESS              0x00000  // Should not change
#define FLASH_IMAGE_1_BASE_ADDRESS              0x10000  // ...

#define FLASH_CALIBRATION_BASE_ADDRESS          0x20000  // Keep on 4K blocks
#define FLASH_DEFAULTS_BASE_ADDRESS             0x30000  // ...
#define FLASH_MANUFACTURING_DATA_BASE_ADDRESS   0x31000  // ...
#define FLASH_INSTALLER_DATA_BASE_ADDRESS       0x32000  // ...

// Clock frequencies
#define OSC_FREQUENCY              100000000   // Actual oscillator frequency
#define CPU_FREQUENCY        (OSC_FREQUENCY/2)  // Typical CPU frequency
```

- **version.h** contains the defines for the software ID, name, version information and the configuration bits. The software ID helps Helios identify the project when it connects. The version numbers are used to get the right description of the global variables. The build number may get incremented automatically by the build scripts when changes are made to the project. Other information can be seen in the comments.

```
#define SOFTWARE_ID_VALUE    0x20150501        // This should be unique for
                                               // each project and branch.

#define PROJECT_NAME         "IXP001 Starter"  // Folder for Helios XML
#define PROJECT_ALIAS        "IXC2 Starter"    // Short name to display in
                                               // Helios tabs

#define VERSION_MAJOR   1
#define VERSION_MINOR   0
#define VERSION_BUILD   24

#define MINIMUM_USER_LEVEL   1     // 1 is full control
                                   // peripherals are level 3

#define CONFIGURATION_BITS 0x7FFF  // Fast boot 0xDBFF
                                   // Slow boot 0x7FFF. See Bootloader manual.
```

- **hdlc_configuration.h** is used to define the HDLC_FUNCTIONS_TYPE structure along with the #includes for the global structure variables you want visible in Helios. The order in the structure determines the five most

significant bits of the address. In math form it looks like (position << 27). The AMBA peripherals assume the zeroth position and the flash base address is determined with a #define.

```
typedef struct {
    HDLC_ACCESS VARIABLES_TYPE_NAME;
    HDLC_ACCESS SOFTWARE_INFO_TYPE_NAME;
    HDLC_ACCESS CONSTANTS_TYPE_NAME;
    HDLC_ACCESS STATES_TYPE_NAME;
    HDLC_ACCESS CALIBRATION_DATA_TYPE_NAME;
    HDLC_ACCESS PLOTTING_DATA_TYPE_NAME;
    HDLC_ACCESS PLOTTING_TYPE_NAME;
    HDLC_ACCESS MANUFACTURING_DATA_TYPE_NAME;
    HDLC_ACCESS HDLC_COMMUNICATION_TYPE_NAME;
} HDLC_FUNCTIONS_TYPE;
```

The HDLC_FUNCTIONS_TYPE structure is a group of read and write function pointers called HDLC_ACCESS which are used to access the values through the HDLC packet. This allows each structure to define and limit how the data is accessed. In some cases, the data may not be a structure in RAM memory, but a data structure in the flash.

The HDLC address map for the above structure is displayed below.

| Index | Base Address | Structure |
|---|---|---|
| 0 | 0x00nnnnnn | AMBA Peripherals |
| 1 | 0x08nnnnnn | VARIABLES Structure |
| 2 | 0x10nnnnnn | SOFTWARE_INFO Structure |
| 3 | 0x18nnnnnn | CONSTANTS Structure |
| 4 | 0x20nnnnnn | STATES Structure |
| 5 | 0x28nnnnnn | CALIBRATION_DATA Structure |
| 6 | 0x30nnnnnn | PLOTTING_DATA Structure |
| 7 | 0x38nnnnnn | PLOTTING Structure |
| 8 | 0x40nnnnnn | MANUFACTURING_DATA Structure |
| 9 | 0x48nnnnnn | HDLC_COMMUNICATION Structure |
| 10 | 0x50nnnnnn | |
| | … | |
| 31 | 0xF8nnnnnn | |

- **hdlc_configuration.c** initializes the pointers to the read and write functions which are used to access the data. You can look at the files variables.h and variables.c to see how these functions and structures are defined.

```
HDLC_FUNCTIONS_TYPE HDLC_FUNCTIONS_TYPE_NAME = {
    .VARIABLES_TYPE_NAME.READ = var_hdlc_read,
    .VARIABLES_TYPE_NAME.WRITE = var_hdlc_write,
    .SOFTWARE_INFO_TYPE_NAME.READ = software_info_hdlc_read,
    .SOFTWARE_INFO_TYPE_NAME.WRITE = software_info_hdlc_write,
    .CALIBRATION_DATA_TYPE_NAME.READ = calibration_data_hdlc_read,
    .CALIBRATION_DATA_TYPE_NAME.WRITE = calibration_data_hdlc_write,
    .CONSTANTS_TYPE_NAME.READ = constants_hdlc_read,
    .CONSTANTS_TYPE_NAME.WRITE = constants_hdlc_write,
    …
};
```

The hdlc_poll() function checks the packets and then calls either hdlc_write_access() or hdlc_read_access() which are defined in this file to further decode the proper functions to call.

# Contents

SA4041 Developer's Guide ..... 1
Introduction ..... 1
About the installed software: ..... 1
Recommended Applications: ..... 1
Software Flow and Hardware Connections: ..... 2
Helios ..... 3
Favorites ..... 4
Getting Started ..... 5
Build process: Test the copied project ..... 13
Project Tour: ..... 14
Helios Communication ..... 16
Adding Variables to Helios ..... 17
Variable naming conventions ..... 17
Unit Symbols ..... 18
Precision ..... 19
Special comments ..... 20
Adding new structures ..... 20
More Structures and Functions ..... 20
Appendix A ..... 22
Contents ..... 25

**Solantro Semiconductor Corp**
**Address: 146 Colonnade Rd**
**Suite 200**
**Ottawa, On, Canada**
**K2E 7Y1**

**Phone: (613) 274-0440**
**Fax: (613) 482-4748**
**Email: info@solantro.com**
**Web: www.solantro.com**