

Integrated Vehicle-to-Infrastructure
Prototype (IVP)

Draft V2I Hub Plugin Creation Manual

Prepared by:

Battelle
505 King Avenue
Columbus, Ohio 43201

Submitted to: U.S. Department of Transportation

Client: FHWA Office of Operations Research and Development
Turner-Fairbank Highway Research Center
6300 Georgetown Pike
McLean, VA 22101

Date: July 2016

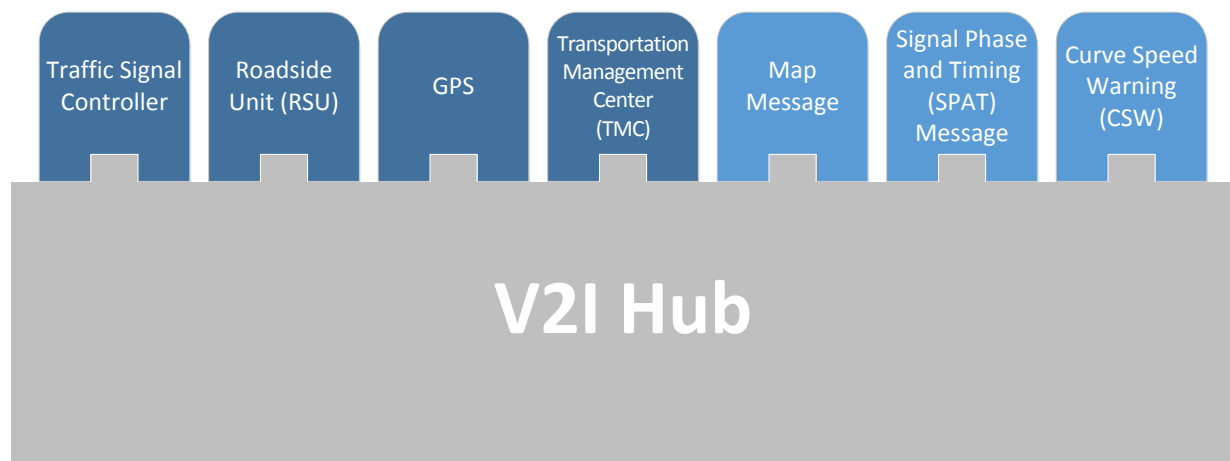


Table of Contents

	Page
Chapter 1. Summary	1
Chapter 2. New Plugin Project	2
2.1. ADDING A NEW PLUGINS PROJECT	2
2.2. UPDATING THE PLUGINS ON THE PORTAL	15
Chapter 3. Receiving Messages from Core	16
Chapter 4. Eclipse Settings	17
Chapter 5. Debugging	19
5.1. CMAKE NOTES	20
Chapter 6. Linking to Common	21
Chapter 7. Finding References/Usages/Examples in the V2I Hub codebase	22

List of Figures

Figure 1. Project Explorer for Eclipse	1
Figure 2. New Project dialog for Eclipse	2
Figure 3. Build Configuration	3
Figure 4. Projects Right click menu	3
Figure 5. Setting C++ include paths for projects	4
Figure 6. Setting C++ flags	4
Figure 7. Project C++ build settings for all configurations	5
Figure 8. Project C++ build settings for debug	6
Figure 9. Build Output	10
Figure 10. Build Output	11
Figure 11. Build Error Output	12
Figure 12. Make Targets	13
Figure 13. Build Outputs	13
Figure 14. Installed Plugins for the Web portal	14
Figure 15. Run Project Button in Eclipse	14
Figure 16. Output from running the project	15
Figure 17. Eclipse Preferences window	17
Figure 18. Sample Main Function for Plugin	19
Figure 19. Eclipse Debug Perspective	19

Chapter 1. Summary

The Integrated Vehicle-to-Infrastructure Prototype (IVP), called V2I Hub, is part of USDOT's Vehicle-to-Infrastructure (V2I) Program and was developed to support jurisdictions in deploying V2I technology by reducing integration issues and enabling use of their existing transportation management hardware and systems. V2I Hub is a software platform that utilizes plugins to translate messages between different devices and run connected vehicle safety applications on roadside equipment.

These instructions are for the creation of a new Plugin to the V2I Hub system. **NOTE:** Integrated V2I Prototype (IVP) Core, V2I Hub and Tmx are used interchangeably in this document.

Assumptions:

- Development operating system is Ubuntu 14.04 64-bit.
- Development environment is already setup and includes GNU C++ compiler and Eclipse IDE.
- Development machine is also used to run IVP core and source code for both IVP Core and IVP API are available.
- Development machine is also used to run web portal.
- Tmx (Transportation Message Xchange) is interchangeable with IVP Core.

Open Eclipse. Make sure the projects for Tmx/Trunk/Core/TmxCore and Tmx/Trunk/Core/TmxApi and Tmx/Trunk/Core/Asn_J2735 are already installed in the workspace. If not, right-click somewhere in the **Project Explorer** space and select **Import...** then under **General**, select **Existing Projects into Workspace** and click **Next**. Use the **Browse** button to select the root directory for Tmx/Trunk/Core/. Ensure that the option **Search for nested projects** is selected, but **Copy projects into workspace** is **NOT** selected. Under the projects list, you should see ASN_J2735, TmxApi and TmxCore. Make sure all of those are selected, and click **Finish**.

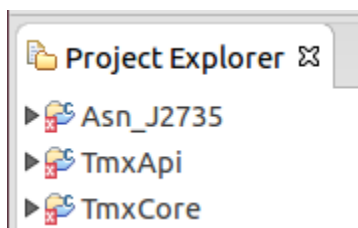


Figure 1. Project Explorer for Eclipse

Until the compiled libraries for the new Tmx versions of these libraries (as opposed to the previous IVP versions) can be installed from the web, these 3 projects need to be able to be built by eclipse – so you'll need to go to the Debug folder of each and cmake it. Run cmake on Asn_J2735 first, then TmxApi, then TmxCore. Cmake is explained later in this document.

Chapter 2. New Plugin Project

2.1. ADDING A NEW PLUGIN PROJECT

- Right-click in the **Project Explorer** and select **New -- C++ Project**. The IVP API does support C projects as well.
- Enter a **Project name**. For this tutorial, our project is called “ExamplePlugin”. Make sure to uncheck ‘Use default location’ and specify the appropriate folder path within your working copy. Select a **Project Type** of “Hello World C++ Project” and a **Toolchain** of “Linux GCC”. Select **Finish**.

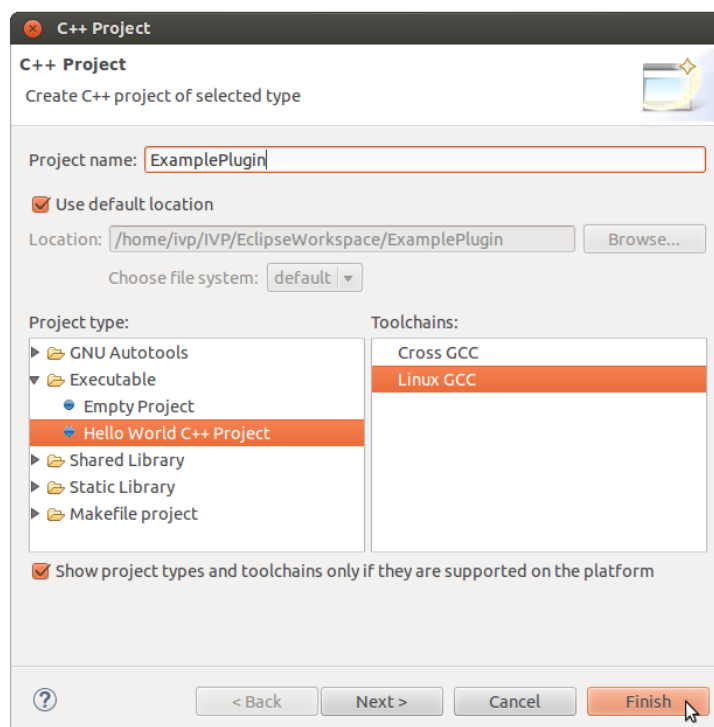


Figure 2. New Project dialog for Eclipse

- Manage Build Configurations
 - Right-click the project name in **Project Explorer**, and selecting **Build Configurations->Manage**. At least **Debug** and **Release** should already be present. Make sure you set **Debug** as the active configuration. In order to add new configurations, such as **ARM-Debug**, select **New**. Creation of the new configuration essentially entails entering the name of the configuration, i.e., **ARM-Debug**, and copying from the existing configuration. Use **Debug** if the new configuration will contain debugging symbols, but **Release** if not.

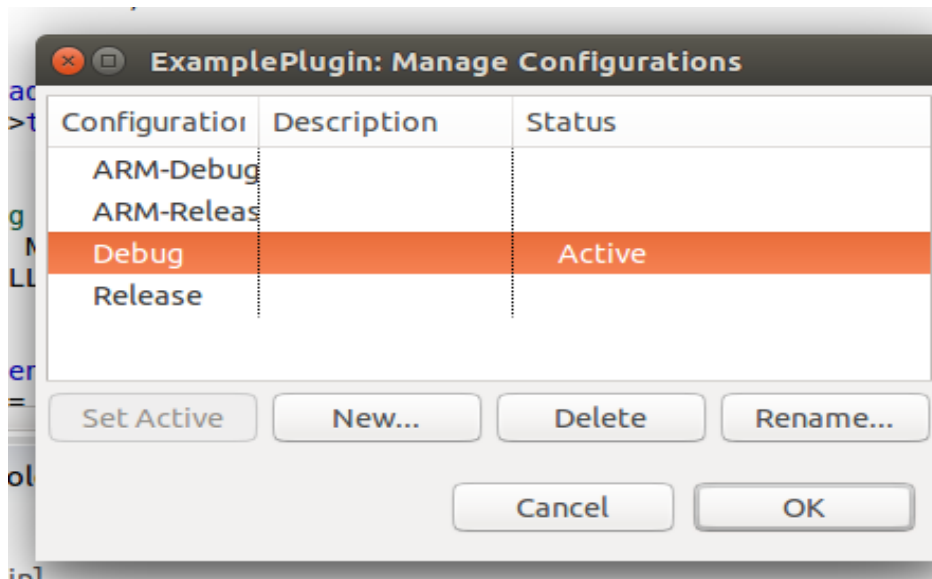


Figure 3. Build Configuration

- For each configuration that will be used in this project, create a corresponding directory by right-clicking on the project name in the **Project Explorer** and selecting **New -> Folder**.
 - Typically, you will want 4 folders in your plugin directory: Debug, Release, ARM-Debug, and ARM-Release.
- Modify project properties. Right-Click the newly created project and select **Properties**.

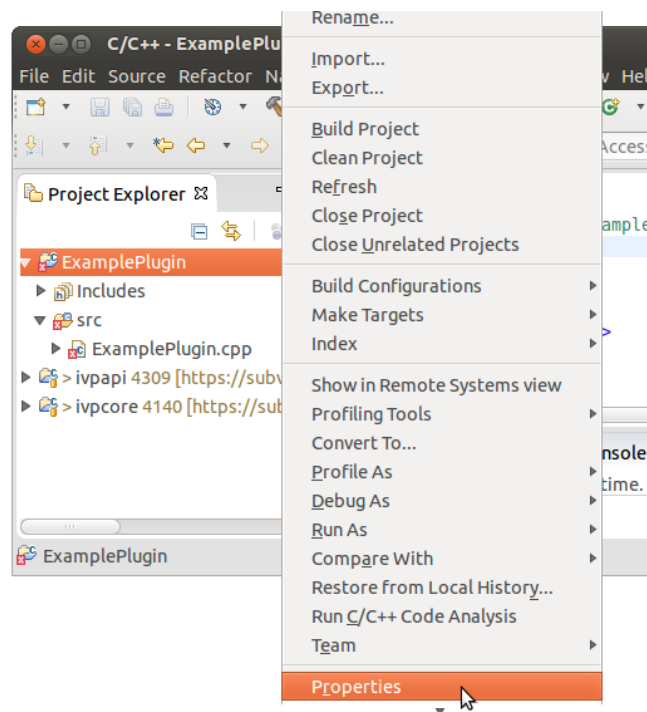


Figure 4. Projects Right click menu

- Add the TmxAPI project folder to **Include paths** for the C++ Compiler as shown below. Click 'Add' then click '**Workspace**' to navigate to the newly added projects. Check '**Add to all configurations**' and '**Is a workspace path**'.

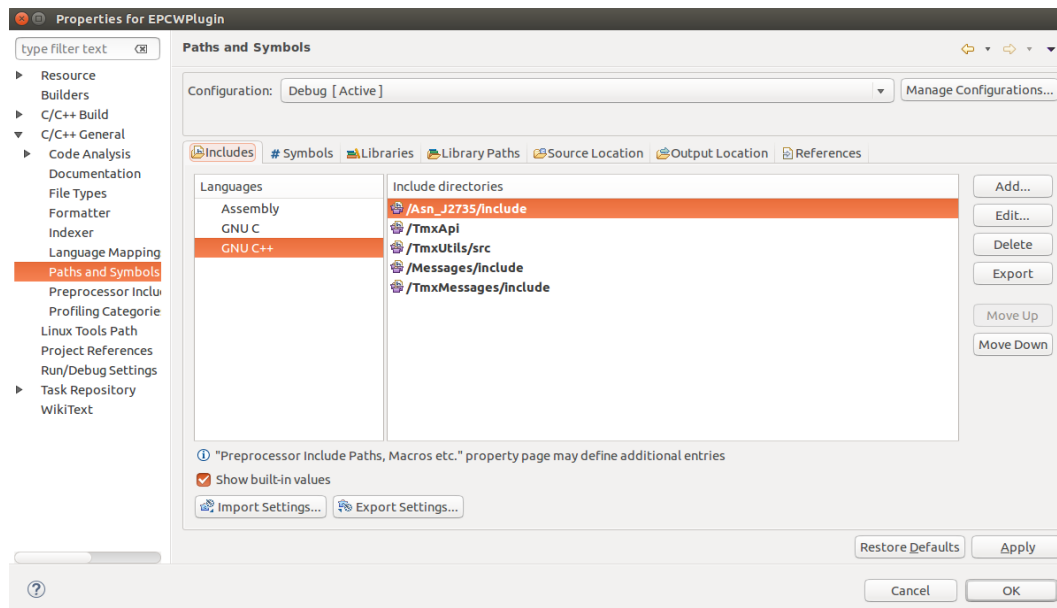


Figure 5. Setting C++ include paths for projects

- Add in a build symbol by switching over to the Symbols tab and include a symbol under the GNU C++ Language. Add in a symbol of `__cplusplus` with a value of `201103L`. *NOTE: There are two underscores before the `cplusplus`.*

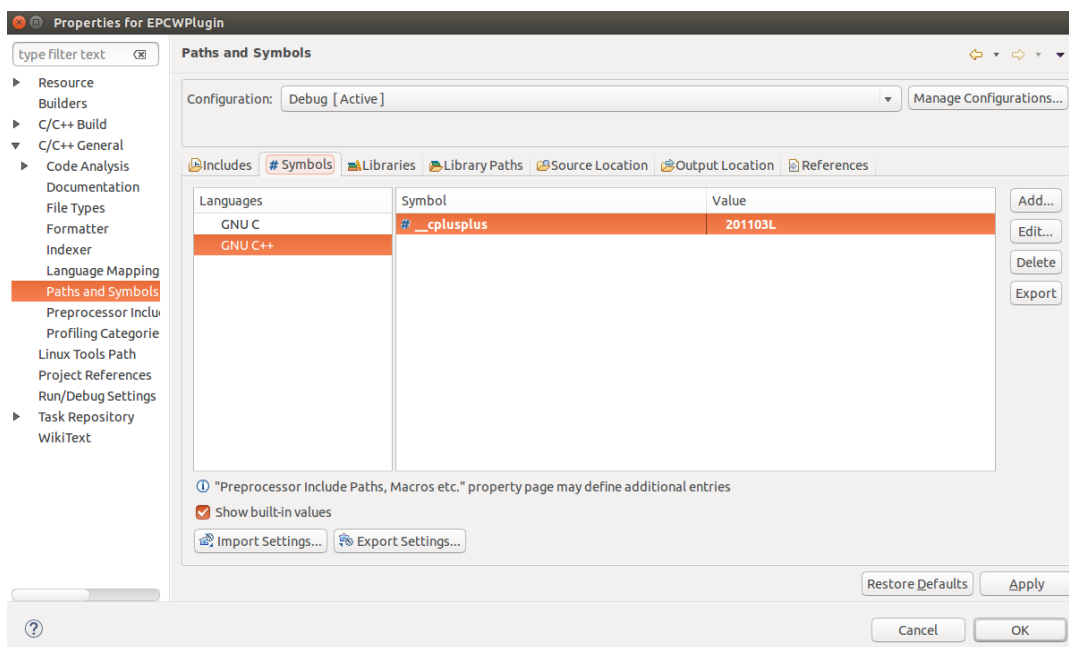


Figure 6. Setting C++ flags

- Modify the setting that manages Makefiles. The default Eclipse build process automatically generates a Makefile; however, we do not use Eclipse to manage the Makefiles so we disable this. Rather, the IVP

projects already generate a Makefile using the **cmake** process. Therefore, for each configuration created, you disable this.

- In the left navigation of Properties, click on **C/C++ Build**. Select the configuration at the top, and click on the **Build settings** tab. Uncheck the box called **Generate Makefiles automatically** and set the **Build directory** to the build configuration directory you built above for that configuration. For example, you can use `${workspace_loc:/ExamplePlugin}/Debug` for the **Debug** config of the **ExamplePlugin**. Make sure to repeat this process for each of the values in the **Configuration** drop down list. Then, click **Ok**.

Note: If build location textbox is read only, or if the Configuration drop down does not seem to work when you want to change the setting for Release etc., 'Ok' to close Properties and open it again, and then the drop down should be ok.

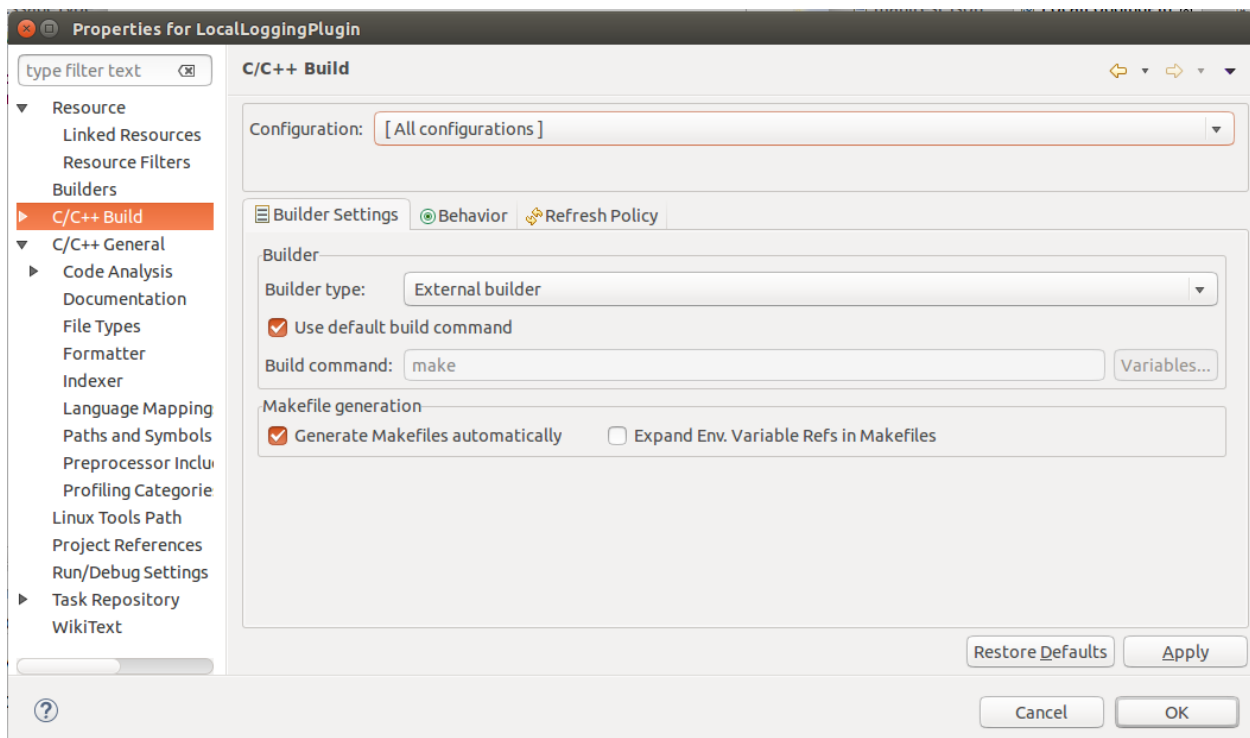


Figure 7. Project C++ build settings for all configurations

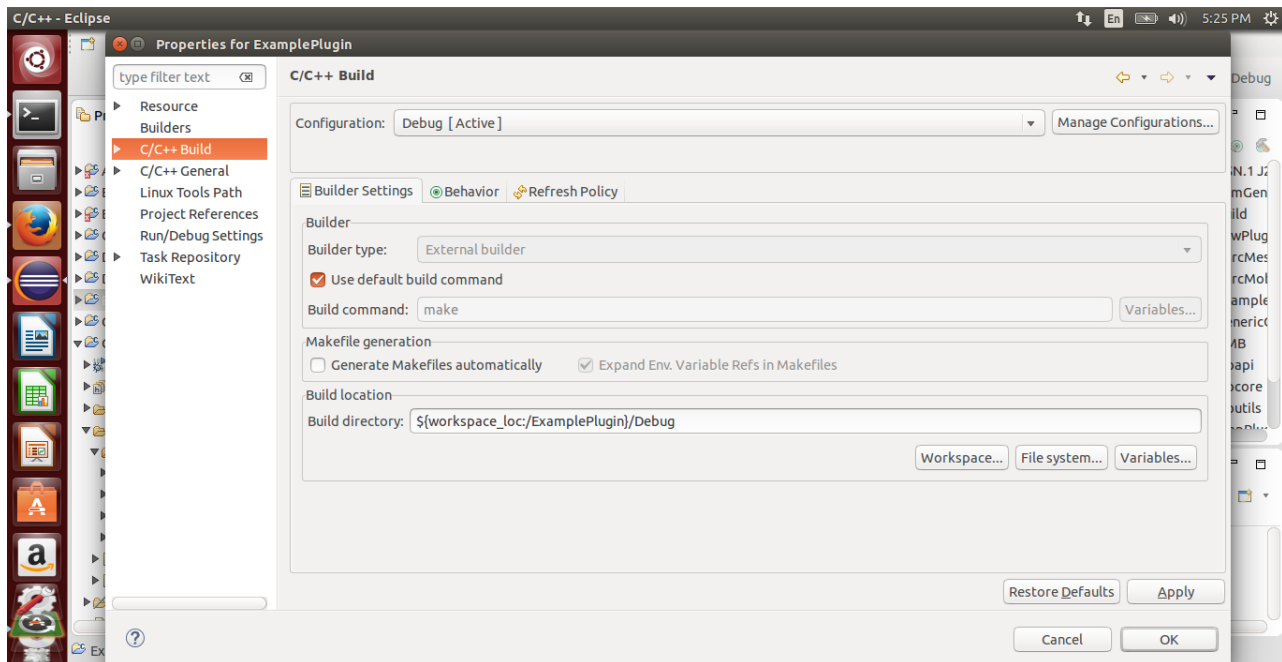


Figure 8. Project C++ build settings for debug

- Modify the cpp source file contents. Delete the contents of the newly created main source code file (e.g., ExamplePlugin.cpp) and replace with the following template code.

```
//=====
// Name: Example.cpp
// Author   :
// Version  :
// Copyright : Your copyright notice
// Description : Hello World in C++, Ansi-style
//=====

#include <iostream>
#include <mutex>

#include <tmx/tmx.h>
#include <tmx/lvpPlugin.h>
#include <tmx/messages/lvpBattelleDsrc.h>
#include <tmx/messages/lvpSignalControllerStatus.h>
#include <tmx/messages/lvpJ2735.h>
#include "PluginClient.h"

#include <LocationMessage.h>

using namespace std;
using namespace ivutils;
using namespace tmx::messages;

namespace ExamplePlugin {

/**
```

```

* <summary>
* Example Plugin. Simulates lat/long, etc. for a vehicle location.
* </summary>
*/
class ExamplePlugin: public PluginClient {
public:
    ExamplePlugin(std::string);
    virtual ~ExamplePlugin();
    int Main();
protected:
    void UpdateConfigSettings();

    // Virtual method overrides.
    void OnConfigChanged(const char *key, const char *value);
    void OnStateChange(IvpPluginState state);
private:
    uint64_t _frequencySetting = 0;
    std::mutex data_lock;

    double latSetting = 0;
    double longSetting = 0;
};

/**
 * Construct a new ExamplePlugin with the given name.
 *
 * @param name The name to give the plugin for identification purposes
 */
ExamplePlugin::ExamplePlugin(string name) :
    PluginClient(name) {
//SubscribeToMessages();
}

ExamplePlugin::~~ExamplePlugin() {
}

void ExamplePlugin::UpdateConfigSettings() {
    // The below code shows an example of how to retrieve a configuration
    // value using the TMXAPI. These should match the manifest.json file.
    // In this case, there is a configuration value named Frequency.
    // All configuration values are retrieved as strings,
    // then must be converted as appropriate.
    GetConfigValue<uint64_t>("Frequency", _frequencySetting, &data_lock);

    GetConfigValue<double>("Latitude", latSetting, &data_lock);
    GetConfigValue<double>("Longitude", longSetting, &data_lock);
}

void ExamplePlugin::OnConfigChanged(const char *key, const char *value) {
    PluginClient::OnConfigChanged(key, value);
    UpdateConfigSettings();
}

void ExamplePlugin::OnStateChange(IvpPluginState state) {
    PluginClient::OnStateChange(state);
}

```

```

        if (state == IvpPluginState_registered) {
            UpdateConfigSettings();
        }
    }
    uint64_t GetMsTimeSinceEpoch() {
        struct timeval tv;
        gettimeofday(&tv, NULL);
        return (uint64_t) ((double) (tv.tv_sec) * 1000
+ (double) (tv.tv_usec) / 1000);
    }

    int ExamplePlugin::Main() {
        PLOG(logINFO) << "Starting plugin.";
        uint64_t lastSendTime = 0;

        while (_plugin->state != IvpPluginState_error) {
            uint64_t frequency = _frequencySetting;
            uint64_t time = GetMsTimeSinceEpoch();
            //Send out the data periodically based on the frequency setting.
            if (frequency > 0 && (time - lastSendTime) > frequency) {
                lastSendTime = time;
                LocationMessage locationMessage;
                //Use methods on the message type to set the values of the fields.
                locationMessage.set_Latitude(latSetting);
                locationMessage.set_Longitude(longSetting);
                //Send the message out through TmxCore.
                BroadcastMessage(locationMessage);
            }
            usleep(50000);
        }

        return (EXIT_SUCCESS);
    }
} /* namespace ExamplePlugin */

int main(int argc, char *argv[]) {
    return run_plugin<ExamplePlugin::ExamplePlugin>("ExamplePlugin", argc, argv);
}

```

- Create a new file called **CMakeLists.txt** under the main directory for the project. This file is used to generate the Makefiles. The basics for a plugin include setting a path to the CMake build scripts, and invoking the configuration created by the **IVP_Plugin_Project.cmake** file. *Note: depending on the file location selected for you project, additional paths to the CMake build scripts may be required.*

```

set(CMAKE_MODULE_PATH "${CMAKE_MODULE_PATH}"
${CMAKE_SOURCE_DIR}/../../../../TMX-OAM/Build/Modules
${CMAKE_SOURCE_DIR}/../../../../TMX-OAM/Build/Modules
$ENV{HOME}/TMX-OAM/Build/Modules
$ENV{TMX_OAM_HOME}/Build/Modules

```

```

)

cmake_minimum_required(VERSION 2.8.12)

project( ExamplePlugin NONE )

include(TMX_Plugin_Project)

find_package(TmxCommon COMPONENTS Asn_J2735 TmxApi TmxMessages)

```

- Create a new file called manifest.json under the main directory for the project. This file is used by cmake (among others). A manifest file is required for the plugin so that it can register properly with the IVP server. Right-click the project name in the **Project Explorer** and select **New – File**. Select the root of the project as the parent folder (e.g., ExamplePlugin) and call the file **manifest.json**. The template source code used for ExamplePlugin.cpp assumes that the file is named manifest.json and that it is at the root of the plugin project. Open the newly created file and paste in the following template text and then save. After the plugin has a manifest file, it can run and connect to the IVP core.

This manifest shows that this plugin generates one type of message (NMEA, Basic) and has 3 configurable settings (Frequency, Latitude, and Longitude).

```

{"name":"Example",
 "description":"Example of Simulates Location data",
 "version":"0.0.1",
 "exeLocation":"/Debug/ExamplePlugin",
 "coreIpAddr":"127.0.0.1",
 "corePort":24601,
 "messageTypes":[
 {
  "type":"NMEA",
  "subtype":"Basic",
  "description":"LocationMessage class holding lat/long etc."
 }
 ],
 "configuration":[
 {
  "key":"Frequency",
  "default":"1000",
  "description":"The frequency to send the Location data in milliseconds."
 },
 {
  "key":"Latitude",
  "default":"0",
  "description":"Latitude."
 },
 {
  "key":"Longitude",
  "default":"0",
  "description":"Longitude."
 }
 ]
}

```

```
}

```

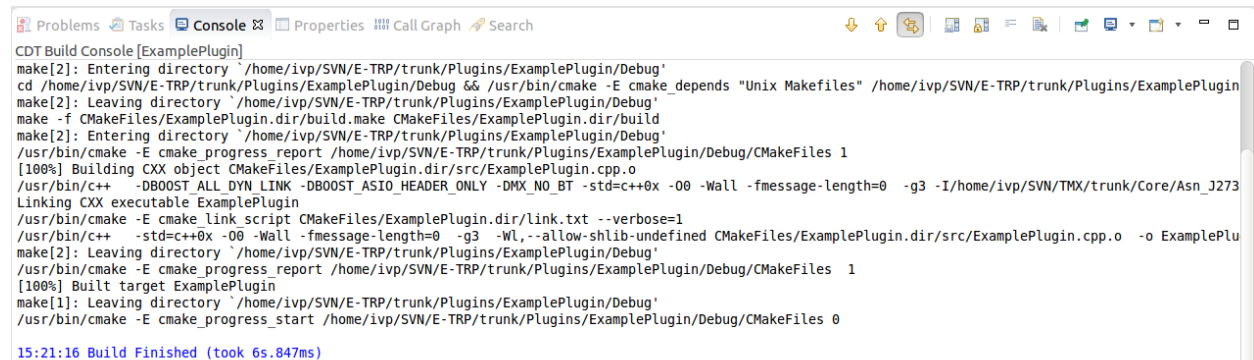
- Run cmake.
 - Open console window and get to the Debug folder of the plugin you are making
-

```
cd PluginsSrc/trunk/ExamplePlugin/Debug
cmake ..
```

Note: If cmakeing outside of the original IVP folder area, cmake cannot find the Libraries directory and gives warning that it's "Unable to locate the Libraries sub-directory." This is okay for Debug and Release. However, ARM builds need this directory for cross compiling (all of the ARM libraries we link against are under Libraries). In these cases, you can add an option to help it find the IVP Libraries:

```
$ cmake -DIPV_LIBRARIES_DIR=<Path> ..
```

- You should repeat the **cmake** steps for each configuration that the project will be used for by navigating to the other Release, etc. subdirectories and running cmake.
- Although the **CMakeLists.txt** file is created under the base directory for the project, it is used multiple times against each build configuration. The default configurations include Debug and Release, which are for the x86_64 target architecture, but there can also be ARM-Debug and ARM-Release for the armhf platform and MIPS-Debug and MIPS-Release for the Arada MIPS platform.
- If you run into errors with cmake/building, it is safe to delete all the contents within the Debug folder, and run cmake again to start fresh.
- Now, you can test that the project builds in Eclipse by right-clicking the project name in the **Project Explorer** and selecting **Build Project**. The Console window should report that the build finished.



```
CDT Build Console [ExamplePlugin]
make[2]: Entering directory '/home/ivp/SVN/E-TRP/trunk/Plugins/ExamplePlugin/Debug'
cd /home/ivp/SVN/E-TRP/trunk/Plugins/ExamplePlugin/Debug && /usr/bin/cmake -E cmake_depends "Unix Makefiles" /home/ivp/SVN/E-TRP/trunk/Plugins/ExamplePlugin
make[2]: Leaving directory '/home/ivp/SVN/E-TRP/trunk/Plugins/ExamplePlugin/Debug'
make -f CMakeFiles/ExamplePlugin.dir/build.make CMakeFiles/ExamplePlugin.dir/build
make[2]: Entering directory '/home/ivp/SVN/E-TRP/trunk/Plugins/ExamplePlugin/Debug'
/usr/bin/cmake -E cmake_progress_report /home/ivp/SVN/E-TRP/trunk/Plugins/ExamplePlugin/Debug/CMakeFiles 1
[100%] Building CXX object CMakeFiles/ExamplePlugin.dir/src/ExamplePlugin.cpp.o
/usr/bin/c++ -DBOOST_ALL_DYN_LINK -DBOOST_ASIO_HEADER_ONLY -DMX_NO_BT -std=c++0x -O0 -Wall -fmessage-length=0 -g3 -I/home/ivp/SVN/TMX/trunk/Core/Asn_J273
Linking CXX executable ExamplePlugin
/usr/bin/cmake -E cmake_link_script CMakeFiles/ExamplePlugin.dir/link.txt --verbose=1
/usr/bin/c++ -std=c++0x -O0 -Wall -fmessage-length=0 -g3 -WL,--allow-shlib-undefined CMakeFiles/ExamplePlugin.dir/src/ExamplePlugin.cpp.o -o ExamplePlu
make[2]: Leaving directory '/home/ivp/SVN/E-TRP/trunk/Plugins/ExamplePlugin/Debug'
/usr/bin/cmake -E cmake_progress_report /home/ivp/SVN/E-TRP/trunk/Plugins/ExamplePlugin/Debug/CMakeFiles 1
[100%] Built target ExamplePlugin
make[1]: Leaving directory '/home/ivp/SVN/E-TRP/trunk/Plugins/ExamplePlugin/Debug'
/usr/bin/cmake -E cmake_progress_start /home/ivp/SVN/E-TRP/trunk/Plugins/ExamplePlugin/Debug/CMakeFiles 0
15:21:16 Build Finished (took 6s.847ms)
```

Figure 9. Build Output

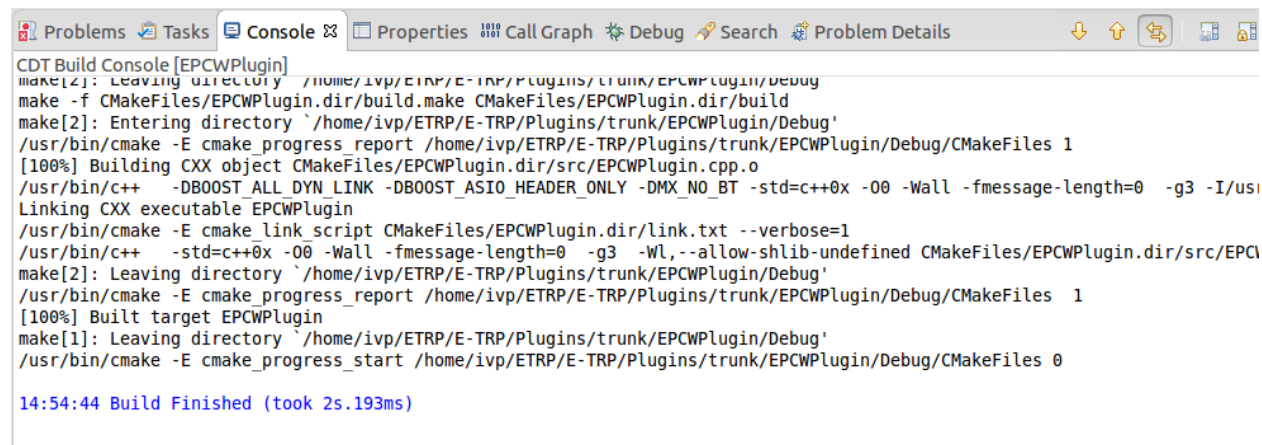
- If your build has issues finding headers/types/or functions:
- Recall that Eclipse does not manage the make file, cmake does. The true cmake build result is displayed in the **Console** tab. The **Problems** tab shows the Eclipse output. These two windows can show different errors. Errors in the **Problems** window may not be preventing the build, but sure make the UI look angry.
- Note that if your build is truly failing, adding (an include), under project properties won't do anything because Eclipse doesn't manage the make file. Cmake creates the makefile. The makefile runs the build.

(If "Automatically create makefiles" was Unchecked, then the added Include directory would appear as a '-I' command line option and the folder would be checked. But Eclipse does not manage the make file, so it does not get added.)

The makefile compile by default only checks include location 'usr/local/include'.

- Right click **project -> Index-> "Freshen All Files"** can also be useful with build errors. Or Right click **project -> Index-> "Rebuild"**. Both in the plugin you are making or in the destination project that the missing thing resides.
- Build succeeding but still have errors:

There may be another situation where the **Console** output displays a successful build (via the Makefile):



```

CDT Build Console [EPCWPlugin]
make[2]: Leaving directory '/home/ivp/ETRP/E-TRP/Plugins/trunk/EPCWPlugin/Debug'
make -f CMakeFiles/EPCWPlugin.dir/build.make CMakeFiles/EPCWPlugin.dir/build
make[2]: Entering directory '/home/ivp/ETRP/E-TRP/Plugins/trunk/EPCWPlugin/Debug'
/usr/bin/cmake -E cmake_progress_report /home/ivp/ETRP/E-TRP/Plugins/trunk/EPCWPlugin/Debug/CMakeFiles 1
[100%] Building CXX object CMakeFiles/EPCWPlugin.dir/src/EPCWPlugin.cpp.o
/usr/bin/c++ -DBoost_ALL_DYN_LINK -DBoost_ASIO_HEADER_ONLY -DMX_NO_BT -std=c++0x -O0 -Wall -fmessage-length=0 -g3 -I/usr/
Linking CXX executable EPCWPlugin
/usr/bin/cmake -E cmake_link_script CMakeFiles/EPCWPlugin.dir/link.txt --verbose=1
/usr/bin/c++ -std=c++0x -O0 -Wall -fmessage-length=0 -g3 -WL,-allow-shlib-undefined CMakeFiles/EPCWPlugin.dir/src/EPCW
make[2]: Leaving directory '/home/ivp/ETRP/E-TRP/Plugins/trunk/EPCWPlugin/Debug'
/usr/bin/cmake -E cmake_progress_report /home/ivp/ETRP/E-TRP/Plugins/trunk/EPCWPlugin/Debug/CMakeFiles 1
[100%] Built target EPCWPlugin
make[1]: Leaving directory '/home/ivp/ETRP/E-TRP/Plugins/trunk/EPCWPlugin/Debug'
/usr/bin/cmake -E cmake_progress_start /home/ivp/ETRP/E-TRP/Plugins/trunk/EPCWPlugin/Debug/CMakeFiles 0

14:54:44 Build Finished (took 2s.193ms)

```

Figure 10. Build Output

But Eclipse is still showing errors under the **Problems** tab:

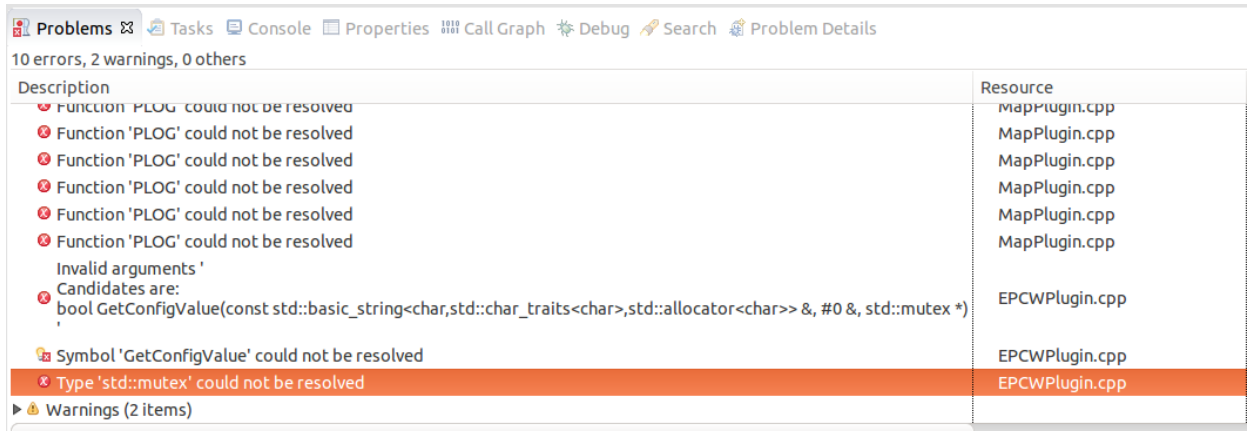


Figure 11. Build Error Output

The Console is the true output, and the exe is still being generated. However, the Eclipse interface is not valid and need to be updated with either Includes or Symbols:

- Fix Eclipse indexer for C++11
 - To fix it, go to **Project Properties --> C/C++ General --> Paths and Symbols --> Symbols --> GNU C++**
 - Overwrite the symbol (i.e., add new symbol): `__cplusplus`
 - With the value: `201103L`
 - Rebuild the index (Right click project -> Index-> Rebuild). If the problem persists, rebuild the index once more.
 - Now that you have a successful build, Start IVP Core. The manifest file is currently configured to talk to IVP core on the local machine using the loopback IP address ("coreIpAddr":"127.0.0.1"). So the IVP core application must first be started on the local machine.

Open a terminal window and browse to the folder containing the compiled ivpcore executable (e.g., `~/IVP/EclipseWorkspace/ivpcore/Debug`). Then type "**sudo ./ivpcore**" to start the core running.

- Create installer package. Before you can run the plugin from within the Eclipse debugger the plugin must first be installed to V2I Hub running on the local machine. You need a packaged ZIP file to do so. However, by default the Eclipse environment will only compile and not package. To add the ability to create the package from Eclipse, right click on the project name under **Project Explorer** and select **Make Targets->Build**. If the **package** target does not yet exist, click Add and type in **package** as the name of the target. Then select the **package** target and click **Build**.

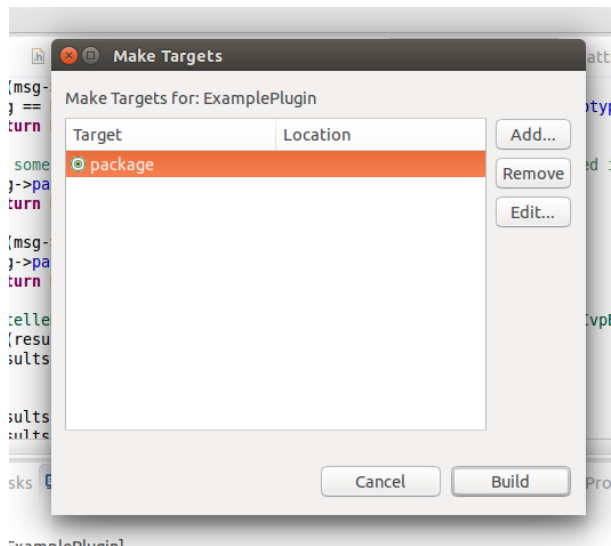


Figure 12. Make Targets

- At the end of the build process, you should see an indicator that the ZIP file was created

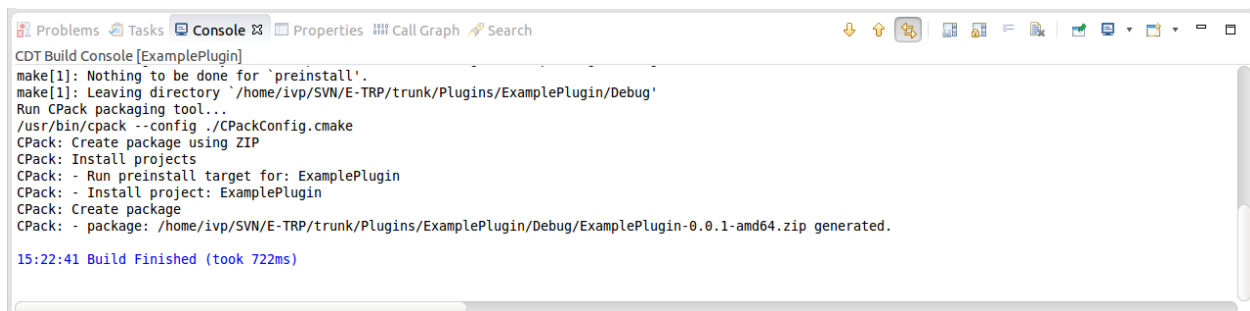


Figure 13. Build Outputs

- That ZIP file should be uploaded to [the local IVP web](#).

Note that the login information can be obtained from Battelle.

Logged in as: **_battelle** (System Administrator) [Log Out](#)

TMX Installed Plugins

Main Menu

- + [Home](#)
- + [Change Password](#)
- + [View System Status](#)
- + [View Event Log](#)
- + [View Message Activity](#)

System Admin Menu

- + [Manage Users](#)
- + [Installed Plugins](#)

Installed Plugins

Plugin Name	Version	Enabled	Max Msg Int	Cmd Line Parameters				
DSRC Message Manager	1.0.0	Disabled	0					
MAP r41	1.1.1	Enabled	5000					

Showing 1 to 2 of 2 records

Copyright (c) 2014 Battelle Memorial Institute. All rights reserved.

Figure 14. Installed Plugins for the Web portal

- Go to **Installed Plugins** page, and add the zip file using the '+' button.
- Click the pencil icon to enable the plugin, but then immediately disable it the same way if you want to run the new plugin in Eclipse.
 - *Note: Max Msg Int is a watchdog timer for plugins that regularly send messages. It counts the seconds since the last time a message was sent, and if another message has not been sent in that amount of seconds, it will reboot the plugin. 0 disables this feature for plugins that are not expected to send regular messages.*
 - Now to Debug in Eclipse, first select ExamplePlugin in the Project Explorer, then click the run icon in the toolbar to start running the plugin.

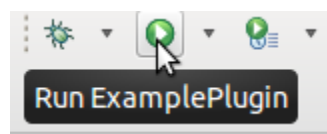


Figure 15. Run Project Button in Eclipse

The **Console** window should show that the plugin has connected and registered.

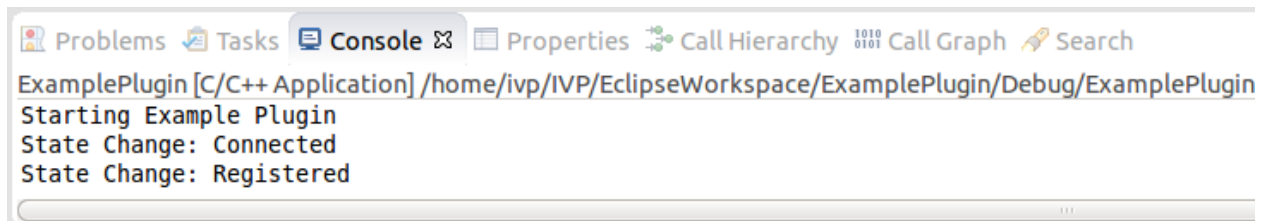


Figure 16. Output from running the project

You can also use the bug icon to start the plugin inside the Eclipse debugger.

2.2. UPDATING THE PLUGINS ON THE PORTAL

If you want to install and run plugins directly out on the Portal, you need to update their zip packages with latest code. The update should not be done when the plugin is running, as I have seen this not reflect the code update and still run old code. Disable the Plugin. Update. Enable the plugin.

Chapter 3. Receiving Messages from Core

The complete example above illustrated how to send messages out through TmxCore. If you'd rather receive messages from TmxCore, you will have to declare a message handler, register to receive those messages, and hook the handler up to the message registration. Excerpts of this are shown below:

```
// Message handler functions. These are registered to be called by the Core when a message
//of that type comes in. Declare these as protected functions.
void handleLocationMessage(LocationMessage &msg, routeable_message &routeableMsg);

...

/**
 * The constructor for the plugin should AddMessageFilter for the types of messages to receive.
 *
 * @param name The name to give the plugin for identification purposes
 */
ExamplePlugin::ExamplePlugin(string name): PluginClient(name)
{
    //Register for the types of messages this plugin should receive. Register a handler
    //function to be called by the Core when that type comes in.
    //We want to listen for location messages
    AddMessageFilter<LocationMessage>(this, &ExamplePlugin::handleLocationMessage);
    //Call SubscribeToMessages once after you've added all the filters you need.
    SubscribeToMessages();
}

/**
 * Called by the Core when messages are received.
 * @param LocationMessage &msg: Contains the contents of the message, strongly typed.
 * @param routeable_message &routeableMsg: Contains header information for the message, including
 * timestamp.
 */
void ExamplePlugin::handleLocationMessage(LocationMessage &msg, routeable_message
&routeableMsg)
{
    PLOG(logINFO) << "Rx LocationMsg. Lat:" << msg.get_Latitude()
<< " Long:" << msg.get_Longitude();

    //Save the timestamp of this message.
    uint64_t MostRecentLocationMsg = routeableMsg.get_timestamp();
}
```

Chapter 4. Eclipse Settings

By default, Eclipse does NOT auto save when you build! Which means your code change is not reflected in the errors unless you clicked save.

To enable auto save for the project you are building: To save resources before manual builds:

1. Click Window > Preferences.
2. Select General > Workspace from the list.
3. On the Workspace page, select the Save automatically before build check box.

*Note if you only do a 'build project' (rather than build All) this **still** only auto saves the **project's** files. If a header was modified in another project it will not be saved and not be reflected in the build results.

If you are used to a Visual Studio environment where when you Debug the UI gets switched over to a 'debug view' automatically, you may wish to set these two "**Open the associated perspective**" settings to '**Always**' under **Window-> Preferences**:

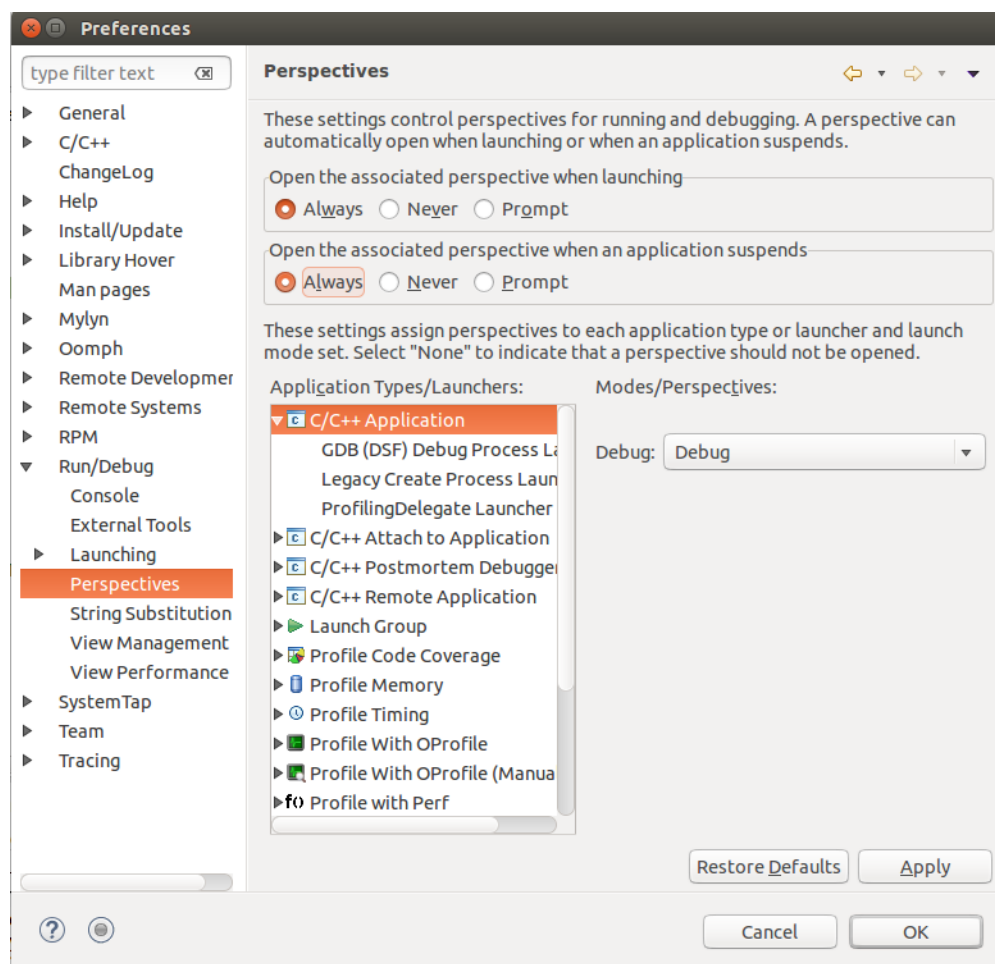


Figure 17. Eclipse Preferences window

That way, when you Debug, the UI will automatically come up with your **Variables** viewer etc. The **Variables** tab is not available under the C++ perspective. Manually switch perspectives in the upper right hand corner.

Chapter 5. Debugging

When repeatedly editing and re-debugging code using the standard green bug Debug icon, it seems very easy for the debugger to get in a bad state. Restarting Eclipse did not seem to help. Often, it is hinted at by the debugger not starting on Main like it normally does (though not always, sometimes it will start ok but not be able to step into lines that are there).

```
int main(int argc, char *argv[])
{
    return run_plugin<EPCWPlugin::EPCWPlugin>("EPCWPlugin", argc, argv);
}
```

Figure 18. Sample Main Function for Plugin

The debug session will *seem* to continue ok, and your breakpoints can still be hit, but the code that seems to be executed is not always what is being executed, nor are your latest changes actually reflected. See example below:

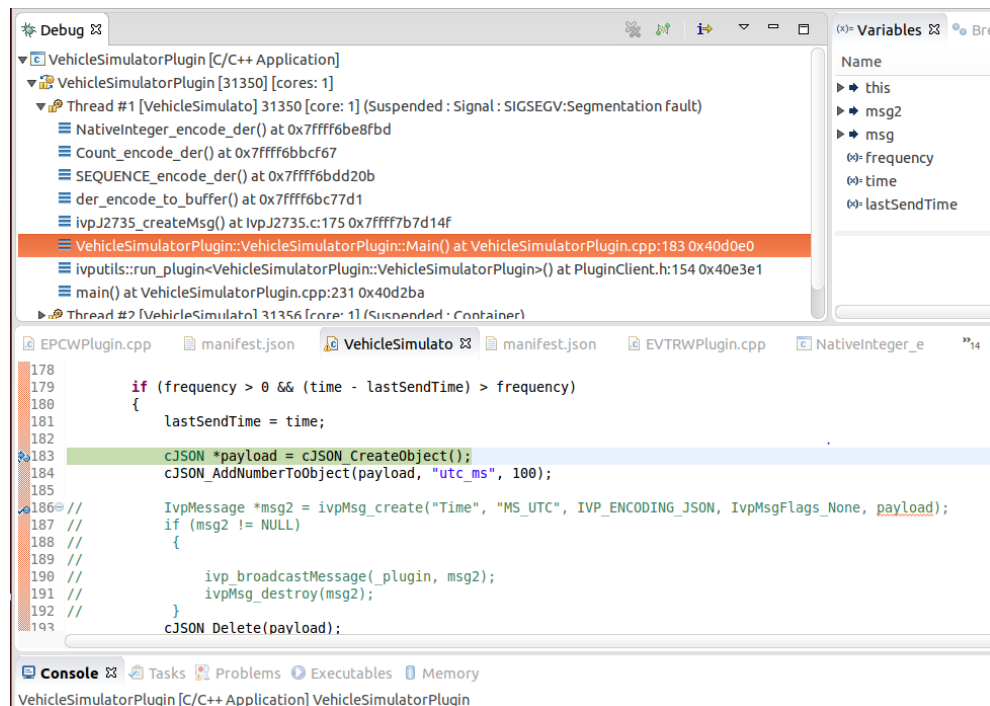


Figure 19. Eclipse Debug Perspective

Where the code was having a segmentation fault on line 183: `cJSON_CreateObject`, but you can see from the stack trace that the error was within `ivpJ2735_createMsg()` that, according to the debugger execution line, was not even being called.

The only time you need to update the package in the website is when you want that new code to be ran when you “Enable” the plugin on the website and want it to run standalone out there.

**Note: Eclipse will let you “run” a plugin (and potentially forget about it) and then let you run that same plugin again – causing strange behavior. If things act up, switch over to the Debug perspective and check to see if you do not have extra processes running.*

The same is true for the Portal. If a plugin is Enabled on the Portal, it will not run correctly in Debug mode. Disable it first.

5.1. CMAKE NOTES

If you add new cpp files to your project you need to rerun cmake or they will not be reflected in the compile!

In order to disable caching used in cmake, delete the entire contents of the Debug folder and then rerun cmake.

Chapter 6. Linking to Common

If your plugin needs access to Common message types, your cmake file needs to be modified so that the linker can find the other pieces.

In order for cmake to tell the linker to link to other Common code, the cmake file must be modified to add a line like:

```
find_package(TmxCommon COMPONENTS Asn_J2735 TmxApi InterPluginSupport
EtrpMessages)
```

and, because InterPluginSupport is a library, the cmake file also needs the line

```
target_link_libraries(${PROJECT_NAME} ${InterPluginSupport_LIBRARY})
```

Two common areas were created under: E-TRP/trunk/Common

Common/Messages:

- Solo header files should be placed under /include. (e.g., Common/Messages/include)

Common/InterPluginSupport:

- Header&cpp combo files should be placed under /src. (e.g., Common/InterPluginSupport/src)
- InterPluginSupport cannot have an include folder. You can only have include OR src, not both.

Example VehicleSimulatorPlugin cmake file that allows linking to Common areas:

```
set(CMAKE_MODULE_PATH "${CMAKE_MODULE_PATH}"
${CMAKE_SOURCE_DIR}/../../../../TMX-OAM/Build/Modules
${CMAKE_SOURCE_DIR}/../../../../TMX-OAM/Build/Modules
$ENV{HOME}/TMX-OAM/Build/Modules
$ENV{TMX_OAM_HOME}/Build/Modules
cmake_minimum_required(VERSION 2.8.12)
project( VehicleSimulatorPlugin NONE )
include(TMX_Plugin_Project)

find_package(TmxCommon COMPONENTS Asn_J2735 TmxApi InterPluginSupport EtrpMessages)
target_link_libraries(${PROJECT_NAME} ${InterPluginSupport_LIBRARY})
```

Chapter 7. Finding References/Usages/Examples in the V2I Hub codebase

Locating references and usages of V2I Hub functions is difficult in the Eclipse environment because most of the plugins and code are not loaded into your workspace. You can you grep in the filesystem, but the UI is not the easiest to peruse.

Recommended:

1. On your windows computer, get the entire V2I HUB codebase from SVN.
2. Install Agent Ransack [Mythicsoft official website](#).
 - a. Use Agent Ransack to find references by pointing ransack at the top IVP folder.
 - b. You may wish to delete/not download Tags folders, as there will be duplicate results.
3. Ransack has an **Options** tab. You can set **Filename** to use “**Regular Expressions**” and then enter “[.]c.{0,2}\$” without quotes to search only .c and .cpp files.
 - a. Or “[.]c.{0,2}|h)\$” without quotes to search only .c and .cpp and .h
4. I did not find a “search whole word” option. If you need this, set **Contents** to search by “**Regular Expressions**” too, and then surround your word with “\W” without quotes.
 - a. E.g., if your whole word was word enter “\Wword\W” without quotes.