IzoT Markup Language (IML) Overview

To create a device for the Industrial Internet of Things (IIoT) you will create a device application that implements the control algorithms for your device and defines your device's interfaces to other devices on a LonTalk/IP or LON network. For example, the application may be a PID controller for a variable air volume (VAV) controller, and the device interface may be network inputs to the device for a temperature setpoint and room temperature input, and a network output for a damper rotation angle. Your device becomes part of a LonTalk/IP or LON network by exchanging data and properties with other devices in the network. The set of datapoints, properties and blocks which you implement to enable such a community of devices to form a distributed peer-to-peer control network is known as the IzoT device interface. The IzoT device interface consists of blocks containing datapoints and properties that define the data published and subscribed by your application, and used to configure your device.

The IzoT Markup Language (IML) provides an easy-to-use and easy-to-learn way of defining your device interface within your C source code. You can use IML with applications that you develop with the IzoT CPM 4200 Wi-Fi SDK or with the IzoT ShortStack SDK.

IML is based on a simple as-if paradigm: when editing your C source code for your program, you declare IzoT device interface items as if your standard C or C++ compiler knew what those meant, and you annotate your declaration with a special tag called the IzoT tag. The IzoT tag starts with ***//@IzoT***, which makes it appear as a comment to your C compiler, but identifies it as the beginning of an IML statement to the IzoT Interface Interpreter. Your C compiler ignores the IzoT tag because it appears within a C comment, but when you build your application a pre-processor called the IzoT Interface Interpreter create a framework that implements your specified device interface and passes the necessary declarations and definitions to the C compiler. The "IzoT" string in the IzoT tag is not case sensitive.

The IzoT Interface Interpreter executes before the C preprocessor so you cannot use preprocessor macro expansion or conditional compilation tools such as **#if** or **#ifdef** to affect IML code.

The IzoT Interface Interpreter converts any field names that conflict with Python reserved words and keywords by appending a single underscore. For example, the ascii member of the **SNVT_str_asc** datapoint type is reported as ascii_.

Example 1

This example implements a single **Variable Air Volume (VAV) Space Comfort Controller (SCC)** block based on the **SFPTsccVAV** profile, including all mandatory components of the device interface defined by the profile including space temperature input and output, a unit status output, setpoints property, and a send heartbeat property. To implement this example, add these two lines of C source code to your application's main C source file:

```
#include "IzoTDev.h"

SFPTsccVAV(vav) vav; //@IzoT block
```

The ***//@IzoT*** block tag specifies a block, with the profile specified by the variable declared to its left. When this code is processed by the IzoT Interface Interpreter, it generates a standard C type (typedef) within the **ShortStackDev.h** file, which your code must include.

The IML declaration for blocks requires a type name modifier, **vav** in this example, because different implementations of the same profile can differ by the implementation of optional profile members and other application-specific details (if permitted by the profile). From your C compiler's point of view, these implementation-specifics require unique C type definitions. The IML language uses the type name modifier to generate unique C type definitions.

Since Example 1 has one block, a Node Object block is not required for this application. Applications that implement multiple blocks require an implementation of the Node Object profile, which provides a common interface for housekeeping diagnostics and maintenance tasks. You can also include a Node Object block if your application consists of a single (other) block.

Example 2

This example implements a **Node Objec**t and an **array of five boilers** based on the **SFPTboilerController** profile.

```
#include "IzoTDev.h"

SFPTnodeObject(node) fbNodeObject; //@IzoT block
SFPTboilerController (boiler) myBoiler [5]; //@IzoT block
```

The C code generated from your IML is contained in two output files, **IzoTDev.h** and **IzoTDev.c**. Do not edit these files. These files are regenerated every time you build your application, and will overwrite any manual edits that you do. You can view both IzoTDev.h and IzotDev.c files to inspect the code generated.

Inspection of the generated data type for your blocks shows how the IzoT Interface Interpreter uses the type name modifier in combination with the profile name to create an unambiguous type definition for this particular block and all its implementation-specific additions or refinements (none in this example).

You must include **IzoTDev.h** with every source file within your application from which you plan to access the members of your IzoT device interface, and you must compile and link your application with **IzoTDev.c**. The IzoT Interface Interpreter also generates IzotInit() and IzotEventPump() functions. You must call the **IzotInit()** function before you access any element of the IzoT device interface or the IzoT Device Stack. While your application is running, you must call **IzotEventPump()** periodically while your application is running to service the IzoT Device Stack.

Example 3

This example adds a **main()** function to Example 2.

```
#include "IzotDev.h"

SFPTnodeObject(node) myNodeObject; //IzoT block
SFPTboilerController(boiler) myboiler[5]; //@IzoT block
int main(void) {
        IzotInit();

        while (1) {
                IzotEventPump();
                your_algorithm();
        }
        return 0;
}
```

You can use IML to implement blocks, device datapoints, device properties, and message tags. You can also specify a number of options which you can use to control details of the generated interface.

The following topics will guide you through the process of creating an application with IML:

- Specifying IML Device Options
- Specifying the Program ID
- Accessing the IzoT Device Interface
- Implementing a Block
- Implementing a Device Datapoint
- Handling a Datapoint Event
- Implementing a Device Property
- Handling a Device Event
- Using a Union Data Type
- Communicating Using Application Messages
- Defining a User Profile or Data Type
- IML Syntax Summary
- IML Errors and Warnings