Introduction to the iris Sensor API

iris INFRARED INTELLIGENT SENSORS

A Tutorial for Beginners

Marvin: I‘ve been talking to the ship‘s computer.

Arthur: And?

Marvin: It hates me.

The Hitchhiker‘s Guide to the Galaxy (The Movie)

Introduction to the iris Sensor API
A Tutorial for Beginners

Revision History

Rev.	Date	Author	Remarks
00	2010-07-10	Christian Klemke	Initial draft including layout and TOC
01	2011-10-01	Christian Klemke	First complete revision
02	2011-10-12	Christian Klemke	Fixes after review by Michael Kirscht
03	2011-10-13	Christian Klemke	Modifications resulting from editorial meeting
04	2011-10-19	Christian Klemke	Corrections after review by Horst Gerland
05	2011-10-25	Christian Klemke	Updates of title graphic and logo
06	2011-11-23	Christian Klemke	Minor fixes, new design
07	2014-01-06	M. Kirscht / C. Klemke	Firmware update chapter based on D5FC files
08	2014-11-26	Christian Klemke	Updated sensor/FA statuses, fixed code sample
09	2015-05-19	Christian Klemke	Added description of start counter
10	2015-08-13	Bernd Schubert	Minor fixes, information on counters updated, information on configuration updated

iris INFRARED INTELLIGENT SENSORS

iris-GmbH
infrared & intelligent sensors
Ostendstrasse 1-14 | 12459 Berlin, Germany
sales@irisgmbh.de

www.irisgmbh.de | www.apc-irma.com

Table of Contents

Introduction....7

The Driver Concept

Driver Responsibilities....10
Driver Mnemonics....11
The Driver Manager....11
Driver Installation and Activation....12
Static Drivers vs. Dynamic Drivers....13
Passing Parameters to Drivers....14
Driver Startup Convenience Method....15

Sensor Addressing

Sensor URLs – a Common Addressing Scheme....16
Device Scan versus Manual Instantiation....17

Working with the DIST500

Sensor Class Hierarchy....20
Base Functionality and the Dist500 Class....21
Subsystem Representation....22
    The DIST500 Class....23
Creating a Dist500 Instance....23
Setting and Getting the Working Mode....24
Activation and Deactivation of the Sensor....25
Setting and Getting the Door State....26
Checking the Sensor and Function Area Statuses....27
Querying Count Results....31
    Height and Object Classification....31
    Category IDs....32
    Counter Basics....34

Counter Usage....35
Counting Related API Parts....36
Accessing the Sensor Configuration....38
The Configuration Manager....38
The Dist500Configuration Class....39
Parameter Groups....40
Reading Parameter Values....41
Performing Firmware Upgrades....42
Communication Firmware vs Application Firmware....42
The Two-Stage Boot Process....43
Parameter Handling During Upgrades....44
Container File Updates....44

UFF Movie Related Classes

UFF Movie Recording....51
UFF Movie Playback....52

Introduction

> *For, usually and fitly, the presence of an introduction is held to imply that there is something of consequence and importance to be introduced.*
>
> *Arthur Machen*

In September 2011, iris celebrated its 20th anniversary. In the course of the two decades since the foundation of the company in 1991, iris APC systems have achieved an excellent reputation in the market, regarding performance and robustness of the equipment no less than in terms of customer service quality. As in all businesses, this kind of success doesn't come for free; it is the result of steady innovative power, diligent work and constant striving for improvement.

It is primarily a simple coincidence in time that the market launch of iris' new flagship product series, IRMA MATRIX, occurred almost in parallel with the corporate anniversary. Nonetheless it proves that even after 20 years, we are not short of innovative ideas or tired of technical progress. Quite the opposite is true: the new DIST500 sensor, which marks the fifth generation of iris APC systems, represents another big step forward in technology. While catching up with the ever-growing requirements of demanding applications, it even offers a number of completely new features which are – at the time of this writing – unique on the market.

Anyone who is familiar with former iris APC systems will notice a major difference between the MATRIX product series and all of its predecessors: those all based on one or more “dumb” sensors which deliver their raw readings to a central piece of hardware, the so-called analyzer. This analyzer contains a sufficiently powerful computing platform and runs clever mathematical and statistical algorithms on the signals received form the sensors and finally outputs the corresponding count results. The analyzer also contains the central power conditioning and supply as well as the interfaces to external systems (usually the on-board computer).

The DIST500 breaks with those former concepts in several ways. First, if you take a look at the MATRIX system architecture, you'll see that the analyzer has vanished. It is not needed any more. During the past years, key industries like IT and mobile communications have pushed the level of integration for electronic components to new frontiers. This development made it possible to integrate the whole functionality formerly provided by the analyzer into the sensor itself; the DIST500 is thus iris' first self-contained, standalone counter.

Second, the DIST500 follows new paths in terms of connectivity. There is an ubiquitous trend in the mass transit industry to overcome the limitations of the interface and bus systems that have been industry standard for many years or even decades (like IBIS, RS-485 or J1708) and establish new standards based on more modern, more flexible and faster communication infrastructure and protocols. As Ethernet and CAN are two of the new “major players” emerging from that modernization process, the DIST500 offers both of them as its primary interfaces, usable in parallel[1].

But good connectivity and ease of integration is not only a question of electrical interfaces. It is also about communication protocols, and the effort it takes to implement all this into the on-board computer software. Actually, the latter aspect is often underestimated. Over the years, iris has helped many customers and system integrators with the integration of our APC products, and a big deal of support was necessary in the field of communication interfaces and protocols, as most of them have their peculiarities.

And there is another obstacle: the existing standard protocols are not suitable to take advantage of the unique feature set of the DIST500. To use this sensor's full potential, it was necessary to create a successor to iris' existing proprietary UIP protocol. This enhanced version is called UIP 2.0 and makes up the foundation of all DIST500 communication, no matter what physical interface is used (Ethernet or CAN). Yet, the DIST500 is an open platform with the ability to support additional protocols of all kinds, including possible future outcomes of the ongoing standardization efforts which iris appreciates and supports by contributing ideas and know-how.

[1] In projects where UIP 1.0, IBIS or J1708 needs to be supported, we use an external gateway interface that acts as “interpreter”. This gateway is based on the proven A21C hardware.

After all, an APC system is usually only one of many components that make up the on-board equipment of modern buses, trams or trains in public transportation. Usually, it does its job silently and unobtrusively, and so should be its integration with the on-board computer firmware. As mentioned above, experience has shown that in practice it is not always as seamless as one would wish for. Having interface and protocol specifications is one thing, but creating a good implementation is often another one. A common wisdom says: “the devil is in the details”, and hardly anyone would deny that. But help is under-way.

Being confident that this step will make the lives of our cooperated software engineers a lot easier, iris took the decision to complement the DIST500 with an SDK including a ready-to-use API and make it available to our partners at no charge. The implementing software engineers thus no longer have to deal with the time-consuming and error-prone task of low-level protocol implementation, but can instead concentrate on the high-level processes. In consequence, there is a chance that development costs and time-to-market can be reduced significantly without having to sacrifice quality. And getting all that for free is a pretty good deal, I suppose.

Now, how is all that connected to this tutorial ? Well, any non-trivial software library needs documentation for the programmers who are supposed to use it. While in rare cases, a pure reference documentation may be sufficient, potential API users quickly get lost if a description of the ideas and concepts is missing. Knowing this, the document at hand was created: a tutorial and a beginner's guide to the iris Sensor API. As it is my first take on that kind of document, there is surely a lot of room for improvement. The same goes for the API itself; despite all planning efforts, only the practical use can show what concepts are good and which ones need to be reworked. In fact, we have already col-lected a number of ideas for improvement, and we invite everyone to participate by send-ing us feedback.

API and tutorial are continuously being worked on, and I expect both to get better with every new release. Despite the existing imperfections, I hope you will find this paper both informative and enjoyable to read.

Christian Klemke, iris-GmbH
Berlin, October 2011

The Driver Concept

> *A careful driver is one who honks his horn when he goes through a red light.*
>
> *Henry Morgan*

Designing and implementing an API sometimes requires a lot of planning. In that phase, it is important to have a complete set of requirements and demands for the final product. Regarding the Sensor API, one goal was to keep the core system light-weight. And it was clear from the beginning that we would want (or need) to support different ways of communication with the sensor in a transparent way. The code for the implementations for these different kinds of communication looks very different in detail, so we came up with a design where these differences are encapsulated in drivers.

In order to support different drivers at the same time (which may be necessary), the need for a central component that is responsible to handle the drivers became apparent. The system architecture was therefore extended by a central driver manager. Having a driver manager plus a number of drivers indeed turned out in practice to be the optimal solution. It helps keeping the system light-weight without imposing limits on functionality.

Driver Responsibilities

The whole idea of a driver is to hide implementation details while maintaining a consistent interface regarding the core functionality. In the context of the Sensor API, drivers play the following roles:

- they enable the usage of the UIP 2.0 protocol on different media (also called transport)
- they allow for a platform-specific implementation of transports
- if necessary, they can be loaded dynamically and parameterized at runtime via an options facility

This still sounds abstract. It is, in fact. Fortunately, to work with the API, you can consider most parts of the API as a “magical black box”: you do not need to understand how it works; you only have to know how you can make it do the things you want to achieve. Nevertheless, it is extremely helpful to internalize at least some basics, some of which are theoretical while most of them will be demonstrated using small snippets of actual C++ code. Let's just move on.

Driver Mnemonics

Every driver comes with a unique string identifier called the driver mnemonic. This mnemonic usually refers to the implemented interface type (e.g. “udp”) and plays an essential role in the Sensor API's internal addressing mechanism (which will be explained in the “Sensor Addressing” chapter starting on page 16). It also serves as parameter to some of the API functions provided by the important DriverManager class which will be described next.

The Driver Manager

Almost every extensible system needs a central mechanism which “ties everything together”. In case of the Sensor API's driver sub-system, this mechanism is the so-called driver manager. Whatever driver related task you have to perform: the driver manager is your friend.

Within the Sensor API, the driver manager is encapsulated in a special class that is implemented after the well-known singleton pattern. A singleton is a class of which exactly one instance exists at runtime. As you might have expected, the driver manager's class name is DriverManager, and it lives in the iris::drivers namespace. You can get a reference to the driver manager at any time by calling its static getInstance() method:

```
using namespace iris::drivers;
DriverManager& manager = DriverManager::getInstance();
```

The DriverManager offers you all kinds of useful functions, some of which shall be highlighted in the subsequent sections.

Driver Installation and Activation

Before you can use a driver, you have to take the following steps:

1. load the driver from an external library
2. prepare it for use by activating it

A driver is loaded by calling the DriverManager's loadDriver() method which takes the file name of the driver library as the first parameter and a additional boolean as the second parameter. This boolean controls whether the driver is automatically activated after loading (or not):

```
Driver& serialDriver = manager.loadDriver(
 “SerialDriver1.dll”, true);
```

As the activation flag defaults to true, it could have been left out in this example. If you choose to only load the driver (by passing false as second parameter), you need to activate manually before you actually use it later on. The startupDriver() method exists for this purpose:

```
Driver& serialDriver =  manager.loadDriver(
 “SerialDriver1.dll”, false);
  ⋮
serialDriver.startupDriver();
```

In the same way, drivers can be deactivated via their shutdownDriver() method if desired:

```
serialDriver.shutdownDriver();
```

For daily use, you can usually stick with a simple loadDriver() call (that is, unless you need to set special options; see “Passing Parameters to Drivers” on page 14 for details regarding driver options).

Beware that you will receive exceptions if any of the above calls fails or if you try to make use of an unactivated driver. Besides, loading a driver twice is an error and will also result in an exception. If you do not know for sure if a certain driver is already loaded or not, you can either catch that exception or (better) use the manager's isMnemonicKnown() method to check if it has already been loaded before:

```
if (!manager.isMnemonicKnown("serial")) {
   manager.loadDriver("SerialDriver1.dll");
}
```

Static Drivers vs. Dynamic Drivers

Anything said above is true for the so-called dynamic drivers which reside inside dynamic libraries (i.e. DLLs/dynlibs). Dynamic drivers are the default inside the Sensor API. However, there is a second option: static drivers. Static drivers are not loaded from an external library at runtime, but are directly compiled into the library.

Static drivers do not need to be loaded with the loadDriver() method. Instead, they have to register themselves with the manager during start-up using the registerDriver() method. At time of this writing, the API contains three static drivers, namely for ICEF, UDP and CAN. In order to instantiate UDP driver statically and register it with the driver manager, use the following code:

```
static iris::drivers::udp::UdpDriver udpDriver;
manager.registerDriver(udpDriver);
```

It is important to know that static drivers are not automatically activated. If you need to use one of them, you first have to get a driver reference and then call startupDriver() on it, like this:

```
Driver& udpDriver = manager.getDriverFor(“udp”);
udpDriver.startupDriver();
```

Passing Parameters to Drivers

In some cases, it is necessary to pass parameters to a driver at runtime. This is done using driver options.

To set a driver option, you can call the driver's setOption() method which takes two strings as arguments: the option name and the option value. Have a look at the following two examples to see how easy this is:

```
udpDriver.setOption(“additionalNetworks”,
 “192.168.3.0:24”);
udpDriver.setOption("additionalHosts",
 "10.3.0.1, 10.4.3.2");
```

Sometimes it is also useful or necessary to ask the driver for an option's current value. Driver provides the getOption() method to do this:

```
string hosts = udpDriver.getOption("additionalHosts");
```

The names of the available options and the structure of the corresponding value strings are driver-specific and can be found in the driver's HTML reference documentation. The example makes use of two options offered by the UDP driver; they extend the scanned IP range by one network and two individual addresses that would not be discovered by the standard scan (which is based on the service computer's network interface settings.)

For completeness, it should be mentioned that any parameter that has been set before can be overwritten by another setOption() call with the same parameter name. To unset an option, the removeOption() method can be called at any time:

```
udpDriver.removeOption(“additionalNetworks”);
```

Likewise, you can check if a drivers supports a specific option and if a certain value is valid:

```
bool unsupported = udpDriver.supportsOption("foobar");
bool invalid = udpDriver.isOptionValid(
 "additionalHosts", "check/Me");
```

Please note: depending on the driver, some or all options may only take effect during the start-up phase of a driver, i.e. they have to be set before the driver is activated. If you need to change such an option for an active driver, you have to first shut it down and then relaunch it with the option changed. Further constraints may apply for certain drivers. However, any restrictions or prerequisites with respect to individual options are indicated in the driver's documentation, so be sure to always read it carefully.

In addition to the online reference, the appendix provides useful information about existing driver parameters.

Driver Startup Convenience Method

As most applications need to support ICEF, UDP and CAN[2], the Driver class provides a static convenience method. A simple call of

```
manager.createAndActivateDefaultDrivers();
```

will try to load and activate the ICEF[3] driver first. If that one fails, it will next attempt to bring up the UDP driver. In any case, it also tries to activate the CAN driver.

The method will not throw any error except if it fails with all three drivers. As this is mostly the desired behavior, we recommend to stick with the convenience method instead of implementing your own startup logic.

---

[2] Out-of-the-box support is only provided for the CAN interfaces manufactured by softing AG. See http://www.softing.com/home/en/industrial-automation/products/can-bus/interface-cards/can/index.php
Other hardware can be used, but requires a small amount of adaptive code to be written. This mainly includes the actual send and receive function for raw CAN frames. The API takes care of the rest.

[3] The ICEF driver is currently deactivated because it has an instability issue that has not been fixed yet

Sensor Addressing

> *A man without an address is a vagabond; a man with two addresses is a libertine.*
>
> *George Bernard Shaw*

Some of the key aspects in designing an object-oriented API is abstraction and generalization. Sometimes this does not only apply to the class hierarchy itself, but also to auxiliary data structures, definition of states or other things. In case of the Sensor API, one of the challenges was to handle to the problem that different transports usually work with completely different internal addresses. The solution we chose is simple yet effective.

Sensor URLs – a Common Addressing Scheme

Anyone who works with internet applications like browsers, email clients or file transfer software is familiar with the notion of URLs. These days, they form a well-known standard for the combination of protocol types and addresses. Furthermore – as URLs are simple ASCII strings – they are human-readable, which is a huge benefit compared to most alternatives.

By offering all these handy features, URLs exactly meet the needs of the device representation problem within the Sensor API: they uniquely identify both a certain protocol and a certain communication entity. Quite consequential, the Sensor API adopts the URL pattern by building its addressing scheme based on simple sensor URLs.

A sensor URL within the API consists of three main components:

1. the one-word driver mnemonic which identifies a certain driver (and therefore a certain communication medium)

2. a separator as used in internet URLs (“://”)

3. a driver-specific address (e.g. an IP or a MAC address), usually including the sensor's UIP sub-address

The following figure shows an example of a "real-world" sensor URL:

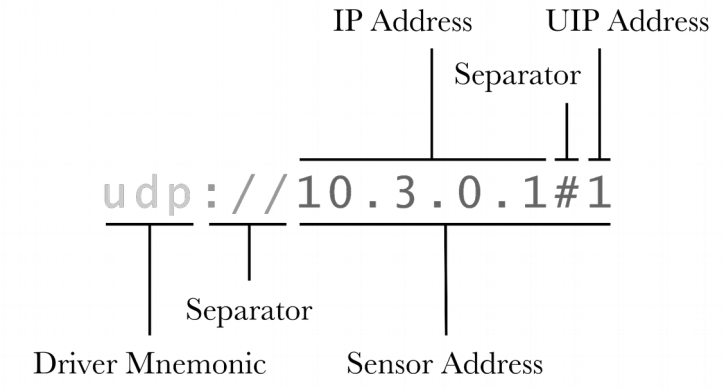

Figure 1: Anatomy of a sensor URL

The `udp` mnemonic in the example belongs to the UIP-over-UDP driver which is an integral part of the Sensor API distribution.

Device Scan versus Manual Instantiation

We have learned from the previous section that communication with any sensor is based on a valid sensor URL. That rises the question how to gain knowledge of that URL. Due to the nature of the problem, there are two possible ways to get the answer:

1. In some scenarios, the application may need to ask the user to choose the desired sensor(s) from a list. For that purpose, the application can leverage the drivers' device scan functionality to assemble information about all available sensors. In a second step it can then present the scan results (or a subset thereof) to the user for selection.

2. In other cases, both driver type and sensor address are already known to the application, either at compile time or at runtime. In such situations, the corresponding URL can easily be composed programmatically by concatenating driver mnemonic, separator and address according to figure 1 (the API even provides static helper methods in the DriverManager class for that purpose).

The Sensor API gives you the freedom to follow either approach, whichever you like better. By definition, the first solution is referred to as device scan approach, while the second one is called manual instantiation.

In either case we can ask the driver manager to create and return a communication object for the URL in question:

```
using namespace iris::drivers;
using namespace std;
string url = "udp://10.3.0.1#1";
DriverManager& manager = DriverManager::getInstance();
Sensor& sensor = manager.getSensorFor(url);
```

As you can see, the example uses a constant URL. The following code gets a list of all UDP-connected sensors:

```
Driver& udpDriver = manager.getDriverFor("udp");
AddressList addrList = udpDriver.performAddressScan();
```

Alternatively, you can initiate the above scan directly via the DriverManger class. A simple call of its scanDevices() method does the trick:

```
addrList = manager.scanDevices("udp");
```

Of course the UDP driver needs to be loaded and activated here, otherwise you will receive an exception. Section "Driver Installation and Activation" on page 12 tells you how to ensure this. Also note that AddressList is nothing but a convenience typedef for a simple list of strings:

```
typedef std::list<std::string> AddressList;
```

This makes it easy to analyze, filter or otherwise process the URLs if that is necessary in your application. Whenever you want to collect the available sensors for all active drivers (not only for a specific one), simply call the scanAllDevices() method of the DriverManager class and you are done:

```
addrList = manager.scanAllDevices();
```

An overview of the different URL structures used by the default drivers included in the default Sensor API distribution is part of the appendix.

Working with the DIST500

> *There are children playing in the streets who could solve some of my top problems in physics, because they have modes of sensory perception that I lost long ago.*
>
> *J. Robert Oppenheimer*

Sensor Class Hierarchy

Most modern electronic devices are highly complex systems made up of different components and subsystems, both in terms of hardware and software. The DIST500 is no exception. As one can imagine, the list of functional units that make up the hardware is noticeable (like power supply, CPU/RAM/ROM, IR lighting, 3D detector chip, several internal buses and IO interfaces, the external CAN and Ethernet interfaces, temperature chip etc.), but it is just as easy to name a huge number of software modules gearing into each other to offer all that nifty functionality offered by a DIST500. This involves modules dealing with low-level hardware components, interrupt handlers and alike as well as mid- and high-level parts like communication protocol drivers up to the main application (including the “heart of our business”, the counting algorithms). All that runs on a real-time kernel, which is a whole software universe itself.

Usually, in any embedded system, different subsets of hardware and/or software components can be identified which cooperate closely in performing certain tasks, thus forming the logical subsystems that make up the final product. The definition of these units can be repeated for different layers from low-level (hardware bound) to high-level (application bound). As this tutorial is about the Sensor API, we will mainly focus on the application layer. After all, we are interested in the DIST500 as automatic passenger (people) counter, so let's see which logical units exist in that context.

Following object-oriented principles, the Sensor API tries to reflect the sensor subsystems in form of corresponding classes. By utilizing C++'s multi-inheritance mechanisms, these classes are re-composed in a second step to the Dist500 class, as shown in the following diagram:

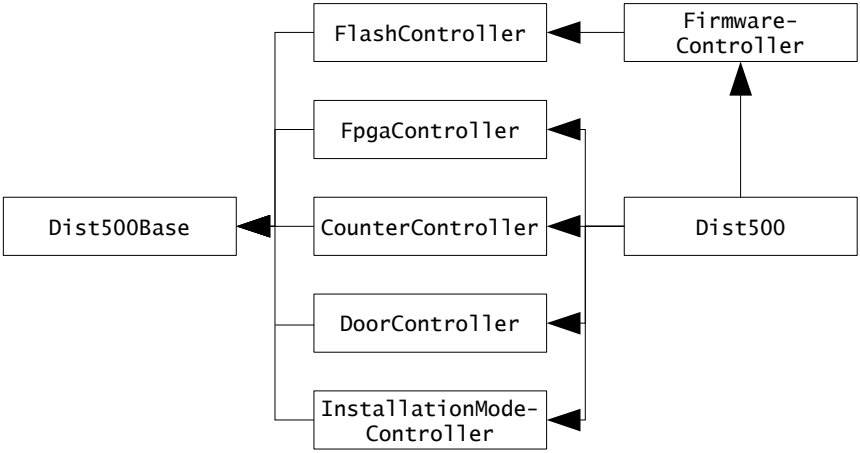

Figure 2: Sensor Class Hierarchy

Base Functionality and the Dist500 Class

There are a few methods that are common to all base classes that actually implement some “practical” functionality. These methods are defined in the Dist500Base class mainly related to transmission and reception of UIP 2.0 messages on using the underlying driver and similar tasks. The vast majority of the Dist500Base functionality is API-internal stuff which you do not need to know about. The only thing I'd like to outline is the fact that in all derived classes like CounterController, Dist500Base is a so-called virtual base class, which is one of the less commonly known features of C++. By declaring a base class virtual, you avoid the replication of members and methods in a multi-inheritance scenario when there are multiple inheritance paths leading from that base class to another one.

That is exactly the case here: Dist500Base is inherited via many intermediate classes (eg. FirmwareController) “into” Dist500, but of course we want all these intermediate classed to share the same members (think of the communication mailbox for incoming UIP 2.0 messages !) and methods. If Dist500Base was not virtual, we would end up with that undesired replication, and the code would not work (or even compile) as intended.

If this is all Greek to you, there is no need to get desperate. Unless you plan to extend the API by implementing your own sensor functionality, you can safely ignore these peculiarities and just enjoy using the existing methods.

Subsystem Representation

In order to better organize the source codes and to reflect the DIST500 software subsystems on the class hierarchy level, the Sensor API defines a number of intermediate classes which are all inherited from Dist500Base as common parent. Each deals with one of the subsystems and allows for interaction with it using a number of corresponding methods. Some of them also contain the handlers for the spontaneous messages the DIST500 sends while the installation mode is active. Here is an overview of the subsystem classes:

- FlashController provides basic Flash memory operations which are not meant for direct use
- FirmwareController contains firmware update functionality based on FlashController
- FpgaController offers access to the internal FPGA, which is only used for special cases in the context of development and metrology tasks
- CounterController implements methods to retrieve count results and reset the counters
- DoorController has methods to get and set the door status
- InstallationModeController allows for control of the installation mode

As before, I will not go into details at that point because usually you work with the Dist500 class directly, not with its base classes. In other words: as a developer using the API, you usually do not have to care about these "intermediate" subsystems classes. The information is primarily presented here because I think it is useful to have a basic understanding of the API structure and the conceptional ideas behind it.

We are now prepared to examine how the Dist500 class is used and how you use it in practice. This includes "daily life" tasks with respect to the regular sensor operation, but also additional tasks in the service and maintenance context.

The DIST500 Class

As shown in the inheritance diagram, all subsystem classes are ultimately reunited in a class named Dist500. It is the central class uses to represent a physical sensor and allows to you interact with this sensor in any meaningful way, using high-level API methods. The creation of an a Dist500 instance is the final step separating us from practical work with our physical sensors.

Creating a Dist500 Instance

During the design phase of the API, it turned out that there will probably be the need to support devices other than the DIST500. In order to keep the basic API (especially the driver part) flexible, we chose to avoid a direct class dependency with respect to the class implementation for a concrete device type, like for example Dist500. This is why the DriverManager does not directly return Dist500 instances, but works with the very general notion of Sensor references instead. Once you have this reference for the desired sensor, the next step is to instantiate a Dist500 object, which then enables you to access the actual sensor functionality, like here:

```
Sensor& sensor = dm.getSensorFor("udp://10.3.0.1#1");
Dist500* dist = new Dist500(sensor);
```

You can think of this instance, dist, as your key to the DIST500 universe. Sticking with the metaphor, the subsequent sections are dealing with the galaxies (read: sensor sub-systems) which reside in it.

Setting and Getting the Working Mode

In order to support special tasks, the DIST500 supports so-called working modes. Depending on the current working mode, some functions may be available or not. The following table shows the currently defined working modes (the list is likely to become extended in the future):

Working Mode	Description
normal_mode	This indicates that no special-purpose working mode is selected
config_and_update_mode	This mode is used during configuration and firmware update tasks
energy_save_mode*	-
signal_record_mode*	-

Table 1: Working Modes

To check the current working mode, call the getWorkingMode() method as presented on the next page:

```
iris::uip::working_mode currentMode;
dist->getWorkingMode(currentMode);
```

A usual, there is a corresponding setter method to change the active working mode, namely setWorkingMode():

* Reserved for future use

```
dist->setWorkingMode(normal_mode);
```

A DIST500 will always run in normal mode after startup. Please note that in systems with an A21C gateway analyzer present, it is necessary to put it into config_and_update_mode for service tasks that involve firmware updates or configuration changes, and to switch back to normal_mode afterwards. This is of course also respected by iris service tools such as the DIST500 configuration utility.

Activation and Deactivation of the Sensor

The DIST500 does not only pave the way to new fields of application, it also overcomes some of the limitations of previous systems. While the availability of accurate information about the door state has traditionally been one of the key factors in gaining high counting accuracy, this requirement does not exist for the DIST500 any more. That is because the high resolution matrix 3D data deliver enough information for the internal track-and-trace algorithms to detect boarding and alighting activity unambiguously even in the absence of external door contact information.

However, there are cases when this is not possible, either because the relevant parts of the scene are covered and thus invisible to the sensor, or because the geometry of an installation is such that door state detection is generally not possible. This is why you can still explicitly tell the sensor about the door state. While in those cases it may be a necessity, doing so is also recommended in cases where it is not mandatory (given the door information is available to the on-board computer, of course).

With respect to activation and deactivation of the sensor, there is another aspect. The DIST500 supports a power-safe mode in which it seizes counting and shuts down the infrared illumination. Supporting this mode of operation is important to reduce power consumption, and it also helps to extend the sensor's lifetime. The latter comes at no surprise if you keep in mind that the ratio between times when counting is actually needed (ie. when people are boarding and alighting at stops) and the remaining time (ie. when the vehicle in on the move) is often 1:3 or even less. It simply wouldn't make sense to keep the sensor switched on all the time.

Controlling sensor activation and deactivation using the API is easy. A sensor is deactivated by calling the stopCounting() method:

```
dist->stopCounting();
```

Usually, you should issue this command when the doors are fully closed and the vehicle has departed from a stop. In many vehicles there is a "departure signal" available which indicates exactly that situation. Whenever it goes active, that is the right moment to call stopCounting().

While the external sensor communication is not affected in any way during power-safe mode, the infrared lighting is deactivated completely. That implies that the 3D sensor chip is temporarily "blind" and therefore the DIST500 cannot further analyze the scene by itself. As a result, you have to tell it to go back to full operation on time, that means shortly before the next stop is reached and doors may open. In order to trigger the transition back to regular operation, call the startCounting() method:

```
dist->startCounting();
```

Please note that if you fail to send the command in time, you may lose counts because the sensor cannot evaluate the scene as long as it is in power-safe mode which means that it may miss relevant actions.

Setting and Getting the Door State

Setting and getting the door state is also straightforward. The Dist500 class inherits the corresponding methods from DoorStateController which defines and implements them. To signal an open door, simply invoke the setDoorState() method as follows:

```
unsigned short fa = 6;
unsigned short door = 2;
dist->setDoorState(fa, door, 100, 100);
```

Note that the function requires the function area address and the door address as first and second parameters. The third and fourth parameter contains the opening percentage for the left and right door wing. Currently, only the "extreme" combinations 100/100 (for fully open) and 0/0 (for fully closed) are supported. That means that in order to signal a closed door, you need to call setDoorState() with both arguments being zero, like here:

```
dist->setDoorState(fa, door, 0, 0);
```

You can also query the current door state from the sensor by use of the corresponding getter method getDoorState():

```
unsigned char leftWing = 0;
unsigned char rightWing = 0;
dist->getDoorState(fa, door, leftWing, rightWing);
if (leftWing == 100 && rightWing == 100) {
   cout << "door is open" << endl;
} else {
   cout << "door is closed" << endl;
}
```

Please note that startCounting() and stopCounting() currently do trigger the same actions within the DIST500. However, this is likely to change in future sensor firmwares, so you should not treat them interchangeably. In other words: use the door methods for actual door information only and the start/stop methods for the general case.

Checking the Sensor and Function Area Statuses

Retrieval of information on the status of an APC system is an indispensable task for any on-board computer firmware. "Status" is quite an abstract term, but that is on purpose because it embraces things of very different nature, like information on sensor sabotage and other self-diagnostic results, but also on inter-sensor communication problems. It also includes simple indication for the availability of new count results. The Dist500 currently provides two general types of status information:

- The sensor status – it is related to a single sensor hardware
- The function area status – is is related to a function area which can be made up from a pair, triple or even bigger tuple of sensors

The API methods to retrieve both are quite similar. Both return the current status in form of a list containing flags which signal certain conditions. The list is empty if there is no such condition. Below is an example which shows the retrieval of the sensor status by means of the getSensorStatuses() method:

```
using namespace iris::dist500;
using namespace iris::uip;
using namespace std;
unsigned short fa = 1;
Dist500::SensorStatusSet sss;
dist500->getSensorStatuses(sss);
if (sss.find(service_mode_active) != sss.end()) {
   cout << "sensor is in service mode" << endl;
}
```

The following table lists the currently defined sensor statuses:

Status	Description
sabotage	Potential sensor sabotage has been detected
config_backup_active	The sensor is operating on parameters from the backup area in Flash because the primary parameter block is corrupted
config_backup_missing	The parameter backup is not present in Flash memory
invalid_config_params	The configuration contains one or more invalid configuration parameters
service_mode_active	The sensor is currently running in service mode
invalid_config	The configuration data contains invalid settings
over_temperature_detected	The maximum temperature was exceeded at least once
over_temperature_active	The maximum temperature is currently exceeded
fpga_restart	The FPGA has been restarted
power_fail	Undervoltage was detected

Table 2: Sensor Statuses

In the same way, getFunctionAreaStatuses() exists to ask the sensor for the status of a certain function area:

```
Dist500::FunctionAreaStatusList fasl;
dist500->getFunctionAreaStatuses(fasl);
for (Dist500::FunctionAreaStatusList::const_iterator
 i = fasl.begin(); i != fasl.end(); ++i) {
   functionAreaStatus fas = *i;
   if (fas.functionAreaStatusBytes ==
    new_counting_result) {
      cout << "fa " << fas.functionAreaNumber
       << " has new counts" << endl;
   }
}
```

Please note that the function area status list contains elements of type functionAreaStatus which in turn contain a number two fields:

```
typedef struct {
   unsigned short functionAreaNumber;
   function_area_status functionAreaStatusBytes;
} functionAreaStatus;
```

The status bytes are listed and explained in the following table:

Status	Description
new_counting_result	There are new count results available in the buffered counters
slave_missing	A master sensor within a sensor group cannot communicate with its slave sensor(s)s
door_clear*	Door area is clear, door can be closed
door_not_clear*	Door area is not clear, door needs to stay open

Table 3: Function Area Statuses

Querying Count Results

Finally we come to the part most of the readers are probably eagerly waiting for: the count results. You have already learned from the list on page 22 that the `Dist500` class inherits the corresponding functionality from the intermediate `CounterController` class. Before we take a closer look at the methods that make up `CounterController`, let's first talk about some peculiarities of the DIST500 sensor with regard to count results.

Height and Object Classification

As you might already have learned from the product description or from the data sheets, the DIST500 offers a unique feature that distinguishes it from competitors on the market: classification of people and objects. That means that the sensor does not simply count anything that is passing its field of view, but has the ability to tell apart people from objects and even to determine an object's type for a number of things that might by of special interest, like suitcases, strollers, wheelchairs or bicycles. For individuals, it can measure their exact size, thus allowing for linkage of person count events to user-definable height classes.

* Reserved for future use

Height classification is a feature often requested in public transportation projects where it serves as an indirect indicator whether the counted individual is a child or an adult. “Indirect” means that of course it is not possible to measure the age of a person; but as there is a correlation between age and body height, it is legitimate to assume that people under a certain size are children whereas people above that size are adults. This isn't exact science, granted. But stochastics provide the mathematical means to deal with such statistical uncertainties, and the final results are sufficiently accurate for most requirements.

Two practical uses have emerged from the body height measurement feature: while in some cases it is a requirement that children be not counted at all (often referred to as height discrimination), in other cases the corresponding counts are of special interest and need to be delivered separate from the adult counts. Both can be easily achieved with the DIST500.

Category IDs

In order to handle stuff as different as people and buggies in a common way in the communication protocol, we have introduced the notion of categories, which is the abstraction of any kind of count the sensor might generate. Every category has a unique ID and some additional meta-data, the most important of which is the category kind. A category's kind is chosen from the following predefined list:

Kind ID	Constant Name	Description
1	person	Person
2	buggy*	Buggy
3	case*	Suitcase or Box
4	dog*	Dog or similar animal
5	bike*	Bicycle

Table 4: List of Pre-Defined Kind IDs

* Reserved for future use

Please note that this list might grow when future firmware gains the ability to recognize new things that are not related to any of the existing kinds.

Depending on the kind of a category, additional attributes may be used to refine the definition of a category. Currently, such a refinement only exists for the “people” kind, namely as the body height interval. To get an idea, please have a look at the table on the next page. It shows how a category definition might look like in practice:

Cat. ID	Kind	Parameters	Description
0	person	parameter1 (min height): 1200, parameter2 (max height): 2500	Adults
1	person	parameter1 (min height): 0, parameter2 (max height): 1200	Children
2	buggy	-	Strollers

Table 5: Example Category Definition

The parameters are stored as four simple integer values; their semantics depends on the actual kind. If necessary, you can request the active category configuration from a DIST500:

```
using namespace iris::dist500;
unsigned short fa = 6;
Dist500::CategoryList cl;
dist->getCountCategories(fa, cl);
```

The category details are represented by the struct category, which has the following definition:

```
typedef struct {
   category_id id;
   category_type descriptionNumber;
   ushort parameter1;
   ushort parameter2;
   ushort parameter3;
   ushort parameter4;
} category;
```

The field descriptionNumber carries the category kind. In case of person, parameter1 and parameter2 hold the lower and upper body height in millimeters. CategoryList is simply an std::list with elements of type category.

Counter Basics

It is important to understand that the DIST500 maintains separate counters for any category it supports. These counters can be read and reset independently if desired. Any operation regarding count results is bound to a category and therefore involves a category ID.

There are two more things need to know about these counters. First, every counter takes care of a pair of values, namely for boarding and alighting people or objects. This is a feature you are probably familiar with. Second, each value exists in two notions: the buffered value and the start value.

Buffered counters and start counters are characterized in the following way:

- After startup, any counter values are zeroed.
- Counting is generally inactive unless explicitly activated by either setting the door state to open via setDoorState() or by calling startCounting(). This also activates the sensors' infrared LEDs. This marks the begin of a door opening interval.
- Any counting event detected by the counting algorithms increases the respective start counter value; it has no effect on the buffered counters.

- Counting is deactivated whenever the door state is set to closed by a setDoorState() call or when stopCounting() is called. Please note that there is usually a short lag before all data is processed and before the LEDs are shut down. This also marks the end of a door opening interval.

- At the end of every door opening interval, the buffered counter values are updated with the corresponding current counts. It is important to understand that the buffered values are not simply replaced by the current counts, but the current counts are added to the buffered values. After the update is done, the new_counting_results[4] flag is set for the function area.

- Both buffered counters and start counters can by queried at any time. The new_counting_results flag is always reset whenever the buffered counters are queried for the respective function area.

Counter Usage

The default approach of count result access is query of buffered counters after end of door opening interval. Characteristics of this approach are as follows:

- You get count result for a stop without any calculations.

- You can use the new_counting_results flag to verify that you query at the right time (strongly recommended).

- There is an additional implementation effort for evaluation of flag new_counting_results.

- Intermediate count values for a long door opening interval are not available.

Demo program “OBC” of IRMA MATRIX SDK demonstrates the default approach.

The alternative approach uses start counter queries. Characteristics of this approach are as follows:

[4] see „Checking the Sensor and Function Area Statuses“ on page 27 for details

- Intermediate count values for a long door opening interval are available.
- Evaluation of flag new_counting_results not necessary.
- You have to calculate the differences to preceding start counter values.
- Counter overrun has to be handled.
- There is no capability to verify right time of query.

Demo program “StartCounter” of IRMA MATRIX SDK shows how to query start counters.

Counting Related API Parts

Now we are prepared to examine the API functions available for the above-mentioned tasks. You have already seen how to query the function area status. So if we know that there are new count results, how can we read them ? Well, that is quite straightforward using the corresponding method. It is called getCountResults()method, and it expects a category ID as parameter. There are two possible ways to use it. If you want a single category only, pass its ID, like here:

```
using namespace iris::dist500;
using iris::uip;
using namespace std;
unsigned short fa = 6;
Dist500::CountList cl;
// buffered adults (category ID 1) only, reset
dist->getCountResults(fa, buffer_counter, true, 0x01,
 cl);
// list only contains one element
category_counts adults = cl.first();
cout << "in " << adults.boarding << " / out " <<
 adults.alighting << endl;
```

In order to access the start counter, simply pass in the start_counter constant as second parameter. The third parameter of getCountResults() is a boolean flag, indicates whether the counter shall be reset or not after the query.

If you want to read the results for all categories at once, pass the wildcard value 0xff as category ID:

```
dist->getCountResults(fa, buffer_counter, true,
   0xff, crl);
for (Dist500::CountList::iterator i = crl.begin();
 i != crl.end(); ++i) {
   category_counts cc = *i;
   cout << "cat " << cc.category_id << ": in " <<
    cc.boarding << " / out " << cc.alighting << endl;
}
```

For both cases, the elements of the list are of the category_counts type which is defined as a structure of three elements:

```
typedef struct {
   unsigned short category_id;
   unsigned short boarding;
   unsigned short alighting;
} category_counts;
```

You can request the sensor to reset a counter at any time independently of getCountsResult() invocation. This is done via the resetCounts() method. It takes only the function area and the buffer type as parameters, e.g.

```
dist->resetCounts(fa, buffer_counter);
```

For development purposes, the ability to programmatically set the counters to user-defined values. If you need to do so, call the setCountResults() method with the list of desired counts:

```
dist->setCountResults(fa, start_counter, false, cl);
```

Here, the boolean parameter indicates whether the new_counting_result flag will be set after the change or not.

Accessing the Sensor Configuration

The DIST500 contains an extensive configuration set. This includes constant information like device number, MAC address and type name, but also a large list of parameters which are meant to be adjustable, either at factory, system integrator or customer level. While many of these parameters (or "configuration values" or "settings") do not need to be touched, others are part of regular tasks, for example installation or bringing to operation. These include the TCP/IP settings, function area and door addresses, information regarding the door physics and a number of similar pieces of information. I suggest to have a look at the DIST500 Configuration Utility, which is part of our DIST500 service software, to get an idea.

The Configuration Manager

The DIST500 contains a firmware part referred to as configuration manager. The idea is to have an advanced mechanism which controls and handles access to any configuration element. "Advanced" means that the configuration manager has a lot of meta information regarding the configuration parameters:

- what data type they have
- what constraints (e.g. upper and lower limits) apply
- what their default value is
- how they are logically grouped
- whether they are used-modifiable or not
- how they are handled during firmware updates

The configuration manager uses these meta data to handle the parameters intelligently even in special situations. These parameter definitions can also be queried from a sensor; the DIST500 Configuration utility for example uses them to apply the correct default values or to enforce existing limits. The corresponding values are not hard-coded into the PC software, but read dynamically from the firmware, which greatly reduces version dependencies between sensor firmware and service tools.

An implementation detail you should be aware of is that the configuration manager does not modify configuration parameters directly in the Flash memory. Instead, it shadows them in RAM and provides means to transfer the data between the two storage areas. By having backup copies in separate Flash banks and adding CRC32 consistency checks, it also makes the parameter storage very stable.

Reading parameters is possible by means of the low-level method getConfigurationParameter(). Before we see how it is used, please note that parameters and parameter groups are internally identified using unique IDs. Any parameter can be addressed using its group ID and Parameter ID (which is only unique within that group). There are also unique symbolic names (string constants) for the groups and their parameters, but usually it is preferable to stick with the IDs because it is more efficient.

The Dist500Configuration Class

The API includes a number of classes which model the complete DIST500 configuration. Using these classes, you can access all parts in an easy and type-safe way. In addition, you can copy the whole configuration to the PC, read and modify parameters locally, and finally write back the whole set using methods. Let's first see how to transfer the full configuration to the PC:

```
using namespace iris::configuration;
using namespace iris::dist500;
Dist500Configuration config;
dist->readConfiguration(config);
```

Parameter Groups

The Dist500Configuration class has a number of methods which return references to the existing parameter groups. The ones for read-only access are used like this:

```
const CommunicationGroup& commGroup = config.
 getCommunicationGroup();
DoorGroup& doorGroup = config.getDoorGroup();
```

The table below lists the currently existing groups:

Class Name	Description
CommunicationGroup	Holds settings regarding communication, eg. IP address and network mask, CAN bus speed etc.
ComponentGroup	Contains the short and the long type name of the sensor and its serial number plus similar information about important internal assemblies
CountingGroup	Stores parameters regarding counting, eg. selection of categories, sensor mounting height and position etc.
DoorGroup	Door specific parameters are stored here; this includes the door address and additional data used by the gateway analyzer (if any), e.g. door contact polarity
FirmwareGroup	Provides information about the firmwares, like name, filenames, version information and so on
FunctionAreaGroup	Holds the function area address(es) and some auxiliary function area data
FunctionAreaDoorGroup	Contains the assignment of door(s) and function area(s)[5]

Table 6: Configuration Parameter Groups by Their Representing Classes

Reading Parameter Values

Each of these groups has getter methods for the parameters it contains. These methods are implemented in a type-safe way, i.e. the method parameters correspond with the actual type of the configuration parameters, like this:

5 Currently, the firmware is limited to exactly one FA/door combination

```
CommunicationGroup::MacAddress mac;
commGroup.getMacAddr(mac);
unsigned short fa;
fa = faGroup.getFunctionArea();
```

Performing Firmware Upgrades

Communication Firmware vs Application Firmware

It is important to understand that the DIST500 relies on two distinct firmwares: a feature-reduced communication firmware (formerly called “bootloader”) and the regular application firmware. The former is executed after startup (or reset) and hands over control to the latter afterwards. That is, as long as some preconditions are met which we will learn about in a moment. For now, please know that both firmwares have a common code base and duplicate some of the functionality, while they differ in major ways:

- The communication firmware only provides a minimum functionality. This includes the full external communication capabilities (CAN and Ethernet), access to the sensor configuration and support for firmware upgrades. Is does not contain any code related to the imaging and thus does not support counting.

- The application has all the elements of the communication firmware, but extends it with the actual application, that is: imaging and counting.

The idea behind the communication firmware is to have a “final line of defense” for the startup process by providing a fallback solution for situations where the application firmware is faulty or even invalid as a result of a former firmware upgrade gone wrong. The next section will make things clearer.

The Two-Stage Boot Process

To understand the idea behind this dual firmware concept, it is necessary to spend a word or two on the way the booting process is implemented in the DIST500. Experience shows that firmware upgrades in the field are – compared to those under laboratory conditions – not free of risk. There are all kinds of things that can go wrong. For example, the supply voltage may be switched off or interrupted accidentally, someone traps on the network cable and disrupts it from the switch. According to Murphy's Law, one or more of these mishaps are likely to occur.

In order to achieve a maximum robustness against such problems, and in order to keep up at least the serviceability under all circumstances, we have implemented a two-staged boot process into the DIST500 as follows:

- The communication firmware starts up
- It checks if some formal criteria are met regarding the application firmware. This includes a CRC32 check of the firmware in Flash.
- If the check fails, the communication firmware remains active.
- If the check passes, the application firmware is booted.
- As an additional safety measure, there is a counter for resets caused by system crashes (watchdog). If the application fails five times within the first few minutes after start, the communication firmware will stop the “vicious circle” by staying active instead of endlessly trying a regular start when this is obviously failing permanently.

These rules will hopefully ensure the ability to get a defective sensor up and running again by correcting its application firmware or by re-adjusting faulty parameters causing malfunction.

Please note that the communication firmware will usually not be touched after production (read: never in the lifetime of the sensor). Should this really become necessary, iris will inform all affected business partners and distribute the new firmware file among them.

Parameter Handling During Upgrades

The Configuration Manager distinguishes between different kinds of parameters with respect to the way they are handled in the course of a firmware update. While some parameters will never change by nature (eg. type name, MAC address and serial number), others may be overwritten when the new firmware comes with new defaults. Yet other parameters have to persist, eg. the door and function area addresses or the internal parameters used for sensor grouping.

Container File Updates

DIST500 firmware container files are special files that bundle one or more, possibly different kinds of firmware along with an optional set of instructions regarding the modification of configuration parameters. This way, components with interdependencies can be packaged which removes the potential risk of miscombinations compared to manual handling of their contents. Consequently, the SDK offers single methods to apply all contents of a D5FC container to one or more sensors without exposing the user to the different steps and the overall complexity of the mechanisms.

Firmware updates based on D5FC container files are done using the method performContainerUpdate() within the Dist500 class. This method exists in two variants: one for the update of a single sensor, and one for the update of a sensor group. It is important to know that multiple sensors within a group are required to run identical firmware, and their parametrization needs to match, and using the multi-sensor SDK update functions prevents violations of these constraints.

The single sensor variant of performFirmwareContainer() is used as shown in the following code snippet:

```
using namespace iris::dist500;
using namespace std;
FirmwareController::UpdateResultMap resultMap;
std::string d5fcFilename =
  „FirmwareContainerFile.d5fc“;
bool result = dist->performContainerUpdate(
  d5fcFilename, resultMap);
```

In case of a sensor group, there is a little more code involved, mainly because the example contains the logic which shows how to identify sensor groups based on a device scan; in practical implementations, you will usually need to implement code which handles the sensors and their relations, eg in order to report failures like sensors completely missing to the higher-level system (ie you usually need to know about the sensors to be expected and other information).

That being said, here comes the example code showing how to call the multi-sensor variant of performContainerUpdate() which has an additional parameter receiving the list of slave sensors. Please note that in this example, only the first sensor group found is updated:

```
using namespace iris::dist500;
using namespace iris::drivers;
using namespace std;
Driver::AddressList addrList =
 dm.scanAllDevices(true);
Driver::Dist500List dist500List;
FirmwareController::UpdateResultMap resultMap;
std::string d5fcFilename =
 „FirmwareContainerFile.d5fc“;
Dist500List remainingGroupSensors;
for (std::list<std::string>::const_iterator
 i = addrList.begin(); i != addrList.end(); ++i) {
   Sensor& s = dm.getSensorFor(*i);
   Dist500* d5 = new iris::dist500::Dist500(s);
   dist500List.push_back(d5);
}
Driver::SensorGroupList groupList =
 Driver::groupSensors(dist500List);
SensorGroup* sensorGroup = groupList.front();
Dist500* distMaster = sensorGroup->getMasterSensor();
  ⋮
for(int index = 0; index <
   sensorGroup->getSlaveCount(); index++) {
   remainingGroupSensors.push_back(
   sensorGroup->getSlaveSensor(index));
}
bool result = distMaster->performContainerUpdate(
 d5fcFilename, remainingGroupSensors, resultMap);
```

The update process happens in several steps. First, the firmware is updated in all sensors. In a second step, configuration parameters are modified according to the instructions in the container file, if any. Between the completion of the firmware update process and the beginning of the parameter handling, a restart of the sensor(s) is triggered. All that happens automatically.

Information regarding the outcome of the update process can be retrieved from the UpdateResultMap instance passed with the method call:

```
typedef std::map<iris::dist500::Dist500*,
 UpdateInformation> UpdateResultMap;
```

This map holds detailed information for every sensor by means of respective instances of UpdateInformation structures:

```
typedef std::list<ParameterUpdateInformation>
   ParameterUpdateInformationList;
typedef struct _UpdateInformation {
   std::list<SensorFirmware> firmwareSkippedList;
   CONTAINER_UPDATE_RESULT updateResultCode;
   ParameterUpdateInformationList
    parameterUpdateInformation;
} UpdateInformation;
```

The field firmwareSkippedList holds a list which indicates firmware which have already be found existing in the sensor. In such cases, the firmware programming is skipped in order to save time, and a SensorFirmware entry is added to the skip list. SensorFirmware is defined as follows:

Symbol	Description
APPLICATION_FIRMWARE	The application firmware has been skipped as the desired version was already present in the sensor
COMMUNICATION_FIRMWARE	The communication firmware has been skipped as the desired version was already present in the sensor
FPGA1_FIRMWARE	The first FPGA firmware has been skipped as the desired version was already present in the sensor
FPGA2_FIRMWARE	The second FPGA firmware has been skipped as the desired version was already present in the sensor

Table 7: Symbols defined by SensorFirmware

The field updateResultCode contains the result of the update process for the respective sensor. It is an enum value of type CONTAINER_UPDATE_RESULT, which means that a value of CONTAINER_UPDATE_PASSED indicates full success of the whole process for the sensor, while any other value signals some kind of error. As there are many different types of potential problems, we have decided not to list them in a table here; please have a look at the reference documentation for a detailed description of all defined result symbols and the corresponding error conditions.

In case there is one or more parameter which could not be written appropriately, information about these parameters is provided collected in the list parameterUpdateInformation containing elements of the structure type named ParameterUpdateInformation:

```
typedef struct _ParameterUpdateInformation {
  unsigned short groupId;
  unsigned short paramId;
  CONTAINER_PARAMETER_RESULT parameterResult;
} ParameterUpdateInformation;
```

For each parameter, it contains the parameter's group ID and parameter ID, along with a result code of type CONTAINER_PARAMETER_RESULT which indicates whether its processing was successful or not (and what kind of problem has occured). Please refer to the table below for a description of the result codes and their meaning:

Result	Description
CONTAINER_WRONG_PARAMETER-_VALUE	A value stored in the container file is invalid
CONTAINER_UNKNOWN_PARAMETER	A referenced parameter is unknown to the SDK
CONTAINER_UNKNOWN_SENSOR-_PARAMETER	A referenced parameter is unknown to the sensor
CONTAINER_PARAMETER_NOT_WRITE	An error occured when writing a parameter
CONTAINER_PARAMETER_WRONG-_UPDATE_STATE	The update type classification of a parameter has prevented its modifcation

Table 8: Container update status codes

Depending on your application, you may want to incorporate some kind of feedback to the user in form of a progress indicator. In order to do so, you first have to define an appropriate callback function that follows the signature defined as UPDATE_CALLBACK:

```
void progressCallback(void* userData,
 iris::dist500::Dist500* sensor,
 CONTAINER_UPDATE_STATE state,
 unsigned int containerStep,
 unsigned int containerTotalSteps,
 double stepPercentage, double totalPercentage) {
   cout << "Step " << containerStep << " of " <<
    containerTotalSteps << ", " <<
    totalPercentage << "% done" << endl;
}
```

The parameter state is of the enum type CONTAINER_UPDATE_STATE and provides information regarding the current status. This status refers to the sensor instance which the sensor parameter points to. In addition, the callback passes on the userData pointer provided when requesting a firmware update, which is done by calling Dist500::performContainerUpdate() method as shown here:

```
bool result = dist->performContainerUpdate(
 d5fcFilename, resultMap, true, 0,
 progressCallback);
```

The parameter userData exists to pass user-definable information to your callback function in case you need it. In the example, it is not used (and therefore 0). Please note that the progressCallback() function must be declared static.

UFF Movie Related Classes

> *There's a lot of great movies that have won the Academy Award, and a lot of great movies that haven't. You just do the best you can.*
>
> *Clint Eastwood*

While UFF movies are unlikely to ever become Hollywood blockbusters, they are still an indispensable tool in the DIST500 software engineering process. This is why the Sensor API offers out-of-the-box support for both recording and playback of UFF movies. Utilizing UFF files has never been easier. Forget about all the low-level pitfalls you might have experienced using the "legacy" UFF DLLs. You can now fully rely on two high-level classes to get the work done: UffRecorder and UffPlayback (hosted under the iris::uff namespace).

Both the UffRecorder and the UffPlayback implementation adhere to a flexible Source/Sink model for data transfer, which makes their use a walk in the park. Recording and playback of sensor signals based on UFF files is easy, as the next sections will demonstrate.

UFF Movie Recording

As UffRecorder is derived from all PictureSink, RegisterValuesSink, DoorStateSink and CountResultSink, it is the perfect counterpart to our well-known Dist500 class. This is of course no coincidence. Telling the API to record data is therefore straightforward:

```
using namespace iris::uff;
UffRecorder* rec = new UffRecorder(“MyRecording.uff”,
 “MyTool”, “1.0”, "myself", "right here");
rec->attachAll(dist);
rec->startRecording();
dist->enableInstallationMode();
  ⋮
dist->disableInstallationMode();
rec->stopRecording();
delete rec;
```

The recording will automatically include images, door status, and register values (temperature and exposure time). Keeping that in mind, I hope you agree that could hardly be any easier.

Please note one thing: the UFF recording only stores door state changes. This means that if the installation mode is already running when you start the recording, you'll miss the initial state that is sent on installation mode startup. This will result in an initial “closed” state in the recording, which may not be correct. In order to avoid that wrong information, you can tell the recorder about the correct state using a special method:

```
unsigned short door = 2;
unsigned char leftWing = 100;
unsigned char rightWing = 100;
rec->setInitialDoorState(door, leftWing, rightWing);
```

In order to be effective, the call to `setInitialDoorState()` must occur before the call to `startRecording()`.

UFF Movie Playback

Playback of an existing UFF recordings is just as easy:

```
UffDecoder* dec = new UffDecoder();
dec->open("MyRecording.uff");
int picCount = dev->getPicureCount();
for (int i = 0; i < picCount; i++) {
   dec->setPictureIndex(i);
}
delete dec;
```

You may wonder why the class is called UffDecoder and not UffPlayer. In fact, both classes exist. While UffDecoder implements the base functionality to read the data from an UFF movie picture-wise (therefore the loop in the example), UffPlayer goes beyond that. It is indeed inherited from UffDecoder and extends it with automatic playback capabilities using a timer thread:

```
UffPlayer upb = new UffPlayer();
upd->open("MyRecording.uff");
upb->setSpeed(2.0);
upd->startPlayback();
   ⋮
udp->waitForEndOfPlayback();
// upb->stopPlayback() can be used to interrupt the
// playback before the end is reached
delete upb;
```

As you can see, starting/stopping the playback

UffPlayer is a descendant of PictureSource, as is Dist500. That makes the two classes interchangeable with respect to picture processing. Whatever code you have developed for online processing of data, you can use it unchanged with offline data (and vice versa). This makes development, optimization and debugging a breeze.