cf/x Script Intelligence Pack

SIP

Script Intelligence Pack

for Unity

v. 1.3

BETA / RELEASE CANDIDATE SOFTWARE WARNING

THIS SOFTWARE PACKAGE IS STILL UNDER DEVELOPMENT. AT BETA STAGE, ALL SOFTWARE IS FEATURES-COMPLETE, BUT MAY CONTAIN SHOW-STOPPING BUGS. DOCUMENTATION MAY CONTAIN REFERENCES TO FEATURES THAT NO LONGER EXIST, ARE PUSHED TO A LATER RELEASE, OR NEVER EXISTED.

ADDITIONALLY, DOCUMENTATION MAY BE INCOMPLETE AND/OR INCORRECT (ACTUALLY, THIS MAY ALSO APPLY TO FINISHED TITLES, BUT WE DIGRESS).

SHOULD YOU FIND INACCURACIES, ERRORS, OR PARTS THAT ARE UNCLEAR, PLEASE LET US KNOW.

IF YOU FIND BUGS IN THE SOFTWARE ITSELF, PLEASE

- DON'T PANIC
- SEND US AN EMAIL AT SUPPORT@CFXSOFTWARE.COM
  AND WE'LL GET BACK TO YOU AS SOON AS POSSIBLE – EITHER WITH A PATCH OR WITH SOME MORE QUESTIONS TO GET TO THE BOTTOM OF THE ISSUE.
- DON'T USE THE ASSET STORE FOR SUPPORT REQUESTS. WE WON'T SEE YOUR POST IN TIME, NOR CAN WE RESPOND ADEQUATELY THERE. PLUS, PEOPLE WHO POST SUPPORT QUESTIONS IN THE REVIEW SECTION LOOK LIKE IDIOTS.

IF YOU HAVE FEATURE REQUESTS, EMAIL THEM TO US. WE **LOVE** FEATURE REQUESTS. WE JUST DON'T PROMISE THAT WE CAN IMPLEMENT THEM. IF YOU ARE THE FIRST TO REQUEST A FEATURE THAT WE IMPLEMENT LATER, YOU WILL RECEIVE AN APPROPRIATE NOD IN THE DOCUMENTATION.

ALSO: YOU ARE BRAVE TO USE BETA SOFTWARE. THANK YOU!

Table Of Contents

Welcome to cf/x Script Intelligence Pack (“SIP”)

Thank you for choosing cf/x Script Intelligence Pack! We hope that you enjoy using this Unity extension, and that it enables you to do the things that you envision faster, more comfortably, and of course, successfully. This document gives you a detailed description of cf/x Script Intelligence Pack’s features and how to use them in your projects.

About cf/x Script Intelligence Pack

cf/x Script Intelligence Pack provides essential functionality that can radically simplify your scripts in Unity – no matter if you are a beginning or advanced scripter. Under the hood, it is a high-performance distributed event handling framework that allows you to quickly and easily exchange information between your scripts and across the network with minimum effort.

For most people, though, it’s just a prefab that you drop into your scene and a *single word* that you change in your script to gain access to one of the most powerful – some say essential – programming features.

cf/x Script Intelligence Pack (SIP) is a collection of prefabs and script extensions that drastically simplify the way you exchange information with other scripts: It allows your scripts to talk to each other – even across the network. You can easily add this capability to your own, and third-party assets. Using SIP can make otherwise complex tasks such as integrating third-party assets, sending notifications across the network, synchronizing multiple object's actions or gathering information about an unknown number of objects simple and – dare we say? – fun. It enables new critical functionality that can multiply your coding ability and efficiency, helps clean up your code, and can make debugging your games much easier. SIP contains powerful Prefabs that can add this capability to third-party assets just by dropping them on the object – a much cleaner solution than adding scripts as components to an object in Editor.

SIP requires a beginner's level of scripting expertise to use, and *greatly* increases in usefulness with your experience. To be able to use the pack you must know how to add a prefab to an object, and have a general understanding of what the Update() method in a script is for. If your scripting know-how extends to variables and methods, buckle up: you're in for a great ride!

Backbone of the pack is a tiny object that you drop into your scene: the cf/x Notification Manager Prefab. It facilitates an incredibly useful feature: the ability of your scripts to talk to each other – even if they don't know anything about the script they are talking to. Your scripts gain this ability by changing a single word in their very first line – from there on you have access to powerful features that we have engineered into a pattern that should be very familiar to you:

- Implement an OnNotification() method in your script, and that method is invoked whenever another script wants to talk to it
- tell SIP when you want to be notified by invoking “subscribeTo()”

- to talk to other scripts, invoke "SendNotification()".

The End. That's why we used the word "engineered", above - not it's lesser brethren "designed".

In other words, the notification feature acts very much like a radio station that allows other scripts to tune in to receive important information and synchronize their actions with it. Building on that functionality we then added the slightly more complex, but radically more powerful “Query” feature. It functions like a two-way radio in groups: instead of simply broadcasting information, it allows everyone to respond. This allows your scripts to broadcast a question, and receive answers from all scripts that have something to say. With this, you can easily implement otherwise complex functionality like Multi-Tags (which we already did for you - for both convenience and educational purposes – it's one of the many Prefabs that come with SIP). Now, to curb your enthusiasm: with great scripting power comes a small price: to use Queries effectively, you must know what a List<T> variable type is and how to use it.

SIP is also network-enabled, allowing you to broadcast notifications across the net to all connected clients at no additional cost. Since multiplayer/networking in Unity requires some intermediate to advanced knowledge, all you need to know is that we’ve got you covered should you get to that stage. When you install the standard SIP notification manager prefab in your scene, you already have everything you need to later add networked notifications.

Speaking of Prefabs, we have included a host of Prefabs that you can attach to existing assets that enable them to respond to notifications and queries – without you ever having to write a single line of code. Better, our scripts attach themselves to your objects at runtime only (as opposed to using Unity Editor to attach a script as a Component). This makes integration *much* cleaner, and easier to manage. You no longer must hunt through hundreds of objects to see if you added a script – the scene hierarchy always shows you any notification or query prefab you added. Using the ability to talk to each other, we have built a number of prefabs for you that provide drag-and-drop, for example:

- Pass notifications (messages and other information) to other scripts (e.g. "Detected an Enemy") – without the scripts needing to know *anything* about each other – not even if they exist at all.
- Be notified when an Animation State changes (Enter/Exit/Stay/IK/Moved)
- Be notified when an Audio Source changes state (Started/Finished playing a clip)
- Be notified when an object (e.g. a third-party asset) collides with another object – without needing to change that asset’s code

In short, SIP allows you to easily do things that otherwise are not-easy-indeed, such as:

- Messing with 'delegates', callbacks or 'Actions' (our notification manager encapsulates all that for you)
- Schedule consecutive tasks
- Synchronizing scripts that control activity

- Multi-Tag management
- ‘Role-call’ like behavior when you need to corral multiple objects with specific attributes that can change during runtime
- Clean up ugly, hard to maintain central Update() calls that mix state polling and decision (e.g. looking at variables to change) with execution (e.g. animating a shader)
- Supports per-script *verbose* property that allows intelligent logging
- Per-script behavior control when parent object or script is active/deactive or enabled/disabled.

Given all of its power, SIP is *tiny*, with a core that is carefully engineered to minimize both memory and performance footprint. It uses less space than a single 512x512 texture, and using the notification mechanism is much less expensive (performance-wise) than using Update() or Invoke(). More importantly, using notifications instead of actively observing an actor during Update() not only cleans up your code, it also makes it faster, easier to export to other projects, and more readable.

History: about cf/x and SIP

We are a small software company that was started in 2003 to produce video effects (that’s the “f/x” part in our name) for Apple’s line of video editing tools: iMovie and Final Cut. In 2010 we were incorporated and branched to full software titles, again focusing on image processing and the artist’s work flow. On the MacOS side of the universe we are proud to be publishing one of the most advanced image compositing titles available (cf/x alpha) as well as some of the most popular utilities for photographers both professional and consumer: WatermarkPRO, Perfect Rename and Photo Mosaic to name just a few.

In 2017 we started a project that brought us to Unity, and cf/x Script Intelligence Pack started as a central pillar to do the heavy lifting inside the game’s event architecture. Now it’s available as a separate asset, fully polished and documented.

About This Document

This document teaches you how to use SIP and the various components that accompany it.

Document Conventions

Throughout this document you will find call-outs that contain important information. Please look for the following:

!	An exclamation mark denotes a paragraph that contains important information that you may want to remember when working with cf/x Script Intelligence Pack.

...	A paragraph with the ellipsis symbol next to it holds some interesting information that may help you understand how a feature works, or how it interacts with others.

Installation and Requirements

Requirements

cf/x Script Intelligence Pack requires a Mac or Windows-based system that runs Unity 5.5 or later. It was successfully developed and tested on both platforms. It likely works with earlier versions of Unity, but it was never tested with them, nor will it be supported.

SIP works for all target platforms builds.

Installing cf/x Script Intelligence Pack

Import the asset from the Unity Asset Store.

Getting Help

You can email us directly with any question or comment at

info@cfxsoftware.com

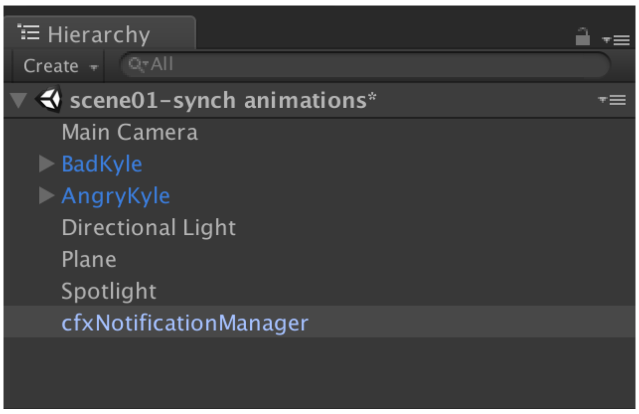
Or you can go to our support page at

www.cfxsoftware.com/support

Quick-Start

Using cf/x Script Intelligence Pack is easy: by changing only one line, your scripts gain an order of magnitude of usability. Using SIP requires three simple steps:

1. Drop the cfxNotificationManager prefab into any scene you wish to use notifications or queries



2. In your script, change one Word: replace "MonoBehaviour" with "cfxNotificationIntegratedReceiver"

   ```
   public class myScript: cfxNotificationIntegratedReceiver
   ```

   (this will also require you prepend ‘public override’ to your Start() method, but more on that later)

3. Implement OnNotification().

You now can communicate with other scripts, broadcast messages, and react when something noteworthy happens.

Let’s try this in a single script to test both:

```
public class test : cfxNotificationIntegratedReceiver {

    public override void Start () {
        base.Start ();
        subscribeTo ("InputReceived");
    }

    void Update () {
        if (Input.GetButton ("Jump")) {
            postNotification ("InputReceived", "Jump Pressed");
        }
    }

    public override void OnNotification (string notificationName,
string theEvent)
    {
```

```
        Debug.Log("I was notified:" + theEvent);
    }
}
```

And here's what the code does:

- We changed the class to be a descendant of cfxNotificationIntegratedReceiver, which gives full access to all notification and query function, and auto-connects to the manager.
- We prepended void Start() with 'public override' and added 'base.Start()' to take advantage of SIP's own start actions. Both are not required in this case, but good practice.
- In Start() we subscribe to receive all notifications that are called "InputReceived".
- During Update() we query Input to see if the Jump button was pressed. If so, we send a Notification with the name "InputReceived" out to anyone who is subscribed to this notification
- Since we are subscribed to that notification, our OnNotification method is invoked. The variable notificationName contains "InputReceived", and theEvent contains "Jump Pressed".

Now, this is entirely unimpressive, as we could have invoked OnNotification from within Update() ourselves, right? Indeed. That is, until you realize, that you can split this into **separate** scripts:

```
public class t_sender: cfxNotificationIntegratedReceiver {

    void Update () {
        if (Input.GetButton ("Jump")) {
            postNotification ("InputReceived", "Jump Pressed");
        }
    }
}
```

and

```
public class t_receiver: cfxNotificationIntegratedReceiver {

    void Start () {
        subscribeTo ("InputReceived");
    }


      public override void OnNotification (string notificationName,
string theEvent)
    {
        Debug.Log("I was notified:" + theEvent);
    }
}
```

that can run independent of each other, know nothing about each other, and **still** cause OnNotification() be invoked. And *now* you get it. Not only will t_receiver's

OnNotification be called when t_sender invokes SendNotification – *so will any other of the possibly hundreds of copies of t_receiver that populate your scenery*! Without you ever having to change a single line of code.

Since most scenes basically consist of many copies of the same script, that’s what makes notification-based programming so effective in Unity.

Part I: The Notification Manager

Introduction

The Marathon Run

Notifications are a simple concept, in programming as in life. The idea is that when something important happens, you inform others through a pre-arranged method: Smoke signals, Flags, Flares, Riders or Runners – humans have been doing it since forever. The Marathon Run is a sporting event that recounts an ancient notification event – when some 2500 years ago a Greek runner ran the 40 kilometers from Marathon (Μαραθών) to Athens to warn the city-state about an impending invasion.

Notifications in Unity

In programming, event-based structures are common, and Unity implements such a pattern with the `Update()`, `Awake()` or `OnActivate()` methods in MonoBehavior.

Unfortunately, though, Unity's implementation of a notification system is incomplete. For reasons unknown, it lacks an inbuilt method for scripts to send arbitrary notifications, along with a flexible way to receive notifications – a way to tell the system that you want to be notified of user-defined events.

A Common Pattern

A common occurrence in games is that things move, and occasionally collide. Projectiles, for example, be they bullets, spears, plasma charges or lightning spells, sometimes hit their targets. Smart programming would have a projectile simply broadcast a collision into the blue, and all scripts that were interested in this event would receive the notification and act accordingly.

Not so in Unity, where you must have at least one script that retains knowledge about the projectile so it can act. If more scripts need to know about this, they also need to retain knowledge about each other (for example what property to set, or which method to invoke). This makes scripting awkward, and prone to errors resulting from broken connections or stale properties – not to mention scripts trying to invoke a method in a destroyed object.

A Higher Degree of Freedom

A proper notification manager can change that: you attach a tiny notify script to your projectile object, and any interested script (usually all potential targets, or a master script that keeps score) is informed when the projectile hits – without requiring links to other objects. This alone makes your projects simpler to implement and better to maintain.

More importantly, it allows you to separate the event handling parts of your code from those parts that try to determine if something has happened (coding boffins will talk about MVC and whatnot – well, using SIP helps you to better separate the V from the C. Feel the power!).

The upshot is that you now can develop scripts independent of each other, as you no longer need to know what properties they have, nor what methods you need to invoke. All you need to know is what notification they are waiting for, and what information that notification should contain.

Using notifications makes scripting in unity much simpler, more elegant and more efficient.

…and Queries, too

Notifications are a powerful concept, especially in Unity, where your scene consists of many distributed scripts. Just by adding notification capabilities, you effectively double the efficiency potential of your code. That being said, Notifications are essentially only one-way communication. You have a script that yells out a command or signal, and all interested scripts initiate their actions upon receiving the broadcast. This is similar to people listening to a radio.

Taking this one step further is being able to respond to a notification, and SIP provides just that with the Query Manager. The Query manager not only distributes a notification, but allows any script that is interested to reply. The Query manager combines all individual replies, and returns this as a single report to the script that issued the Query. So, if a script queries "Who is still alive", it might receive a list of the five objects that have responded to the query.

Designed for Unity

Technically, the Notification Manager is modelled in part after NextSteps/Cocoas Observer pattern, but our implementation is designed with Unity's unique (dis-) abilities in mind. As such, it does not utilize multithreading (yet), since Unity's API does not respond well to background tasks accessing it. Instead, it manages communication intelligently, and reduces messaging overhead to a minimum, while still providing a lot of convenience like scheduling and automatic filtering of events. The Query Manager that resides on top of the Notification Manger is not usually part of a notification system, but was added as a natural fit for Unity's unique distributed script environment.

We designed SIP specifically for Unity, with the goal of being transparent yet fully accessible. If you aren't technically inclined, all you need to do to gain access to all notification functionality is to change the `MonoBehaviour` word in your script. From there on you can post notifications to all other scripts using `sendNotification()`, and receive Notifications with the `OnNotification()` method. To use Queries, you implement OnQuery() and runQuery().

If you are interested in technicalities, we decided to go a somewhat unusual route: we chose to make SIP a base class for your scripts instead of providing a library. This means that all functionality is available to all your scripts all the time, while only requiring you to change a single word in your script. We implemented the notification and query managers as a MonoBehaviour-based Responder script that implements all notifications under the hood, and by changing your script class from `MonoBehaviour` to our `cfxNotificationIntegratedReceiver` class, you inherit all functionality and capabilities - just by changing that one word.

By 'inheriting' SIP (i.e. exchange `MonoBehaviour` for `cfxNotificationIntegratedReceiver`, your script gains the following:

- Ability to receive notifications
- Ability to send notification
- Convenience functions to filter and process notifications
- Ability to receive queries
- Ability to post (‘run’) queries
- Ability to assign your own responder methods
- Automatically detect scene changes

Your script now automatically connects to the scene's Notification Manager by itself, you do not have to do anything (except dropping the cfxNotificationManager prefab into your scene). Connecting to the Notification Manager, posting notifications or running queries does not carry a performance penalty. Event better, responding to a notification or query is much less performance costly than an `Update`() or (shudder) an `Invoke`() message, and the Notification Manger’s overhead is negligible (but your response to notifications/queries may cost performance, so continue to program wisely).

Across the network

SIP has network ability built in. From the moment you drop the SIP notification manager prefab into your scene, you have everything you need to also support network-wide notification broadcasts. Should you then later decide to upgrade your scene to multiplayer/networking, you don’t have to change a single line in your scrips to receive networked notifications.

We interrupt this program…

So, what are notifications, and why are they useful? Consider the following scenario: you are waiting for the delivery of a gift for your partner. You can

- Wait by the door until the courier arrives *or*
- Continue doing what you do, and have the courier call you when they arrive. When you receive the call, you open the door.

First, note that neither approach will make the delivery arrive faster (even though waiting by the door may make it *seem* longer, somewhat proportionally to how urgently you await delivery). The first one wastes your time, while the second one requires a means of communication by which the courier can tell you that they have arrived. The notification manager is such a device, and we'll soon see how it helps you clean up your code, and improve performance.

Busy Wait

The first approach is implemented by what is called a 'Busy Wait' or 'Polling' method, and it is something that you probably have done many times:
In your `Update()` method, you look for a change in a variable, and only do something when it changes. This is, performance-wise, costly and looks, aesthetically-wise, ugly. It's also difficult to read, as many of your decision-making stuff ends up bunched inside the `Update()` function.

Notification

The second method is also something you have done many times, you just haven't looked at it that way. Whenever you implement an `Update()` method in your script, you are implementing a response to a very specific notification. `OnActivate`(), as another example, is Unity letting your script know ("notifying it"), that the Object has become active. In practice, you probably have wished for some `OnSomething`() notification that allows you to respond to an event that wasn't foreseen by Unity's MonoBehavior. If you ever wanted that, you were wishing for a general notification system – which you now have. Given that notifications are one of the central pillars of modern programming it is anyone's guess why Unity doesn't come with a built-in system.

In general terms (and as implemented here), a notification system is nothing more than to agree on a method to notify (in real-world perhaps a phone, here the notification manager), and how to interpret the notification. As an infamous example, in World War II the Japanese allegedly used the code word "Tora" (Tiger) to notify command that complete surprise was achieved, and the Pearl Harbor attack should commence.

In SIP, we use strings and dictionaries to pass notification data between scripts. You decide yourself how to use this data and how to interpret it; the notification manager's only job is to deliver it reliably. Divorcing notification delivery and content in this way allows you to write scripts that don't know anything about each other except the names of the notification, and how to interpret the information it receives.

Guardians of the scenery

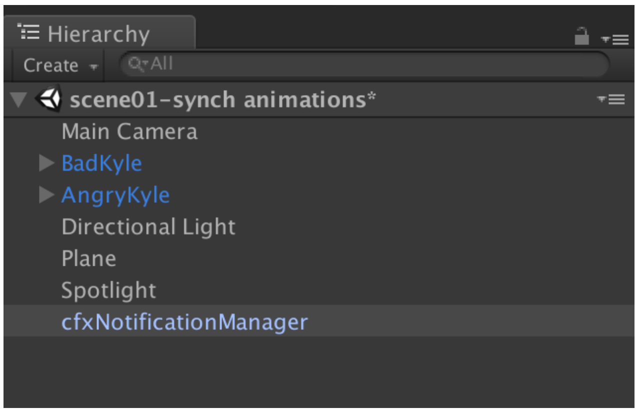
For example, a 'Guardian AI' script does not need to know how many, or what other scripts are running. It knows nothing about the various scripts that control sensors or detectors, or unit behavior. And those scripts know nothing about the guardian object; neither if it exists, nor what properties it has, nor which methods need be invoked. All they know is to send a notification when it’s time, and the Guardian object knows that when the 'Enemy Sighted' notification comes, it must react. This nicely illustrates the versatility you gain when using notifications versus invoking methods and managing objects.

Using the Notification Manager

The notification manager consists of two distinct parts, plus a collection of optional utilities that you can deploy whenever needed. Once you have imported the asset, using the notification manager is surprisingly simple:

Step 1: Deploy the Notification Manager

The first part is simple: you must install the notification manager in your scene. To do so, drag the cfxNotificationManager prefab into your scene.



It is self-configuring, tiny, and consumes only a negligible amount of processing power. It won't even be called on Update(). You should add it to all your scenes just on principle.

Step 2: Empower your Scripts

To communicate with the communication manager we designed a number of classes on top of MonoBehavior. You get all this information basically for free by basing your (existing) scripts on the cfxNotificationIntegratedReceiver instead of MonoBehaviour.

! If you are using your own classes, you can add SIP abilities by looking for the base class that inherits from MonoBehaviour, and change that class. All descendants will then inherit SIP as well.

So how do you do this? All you need to do is look for the 'MonoBehaviour' word at the top of your script, and exchange that with cfxNotificationIntegratedtReceiver. So, if you created a script called "complainAI", look for the following:

```
public class complainAI : MonoBehaviour {
```

and change it so:

```
public class complainAI : cfxNotificationIntegratedReceiver {
```

That's all. Your scripts can now subscribe to, send and receive notifications. We'll show you how in the next chapter.

Step 3: (Optional) use integration Prefabs

You are probably using a host of third party assets that you want to integrate with your project. Luckily, this has just become a lot easier with notifications, and we provide a host of small prefabs that attach themselves to assets, and send configurable notifications when you need them.

We provide drop-in scripts or prefabs to give you notification access to important events such as an animation starting/ending or whenever an audio clip completes playing. We have dedicated a whole chapter to using SIP for easier integration, and we believe you'll be equally parts impressed and happy when you find out just how simple many of your tasks have become.

Step 4: (Optional) Change your Start() message

When you use SIP, you should slightly change your own Start() method to also invoke the Start() method that your script inherits from cfxNotificationIntegratedReceiver as follows (changes in red):

```
public override void Start() {
    // call SIP's Start()
    base.Start ();
    // your own code here
}
```

If you don't do this, a number of SIP's more advanced features like automatic scene change detection, as well as the `whoIs()` and `DelayedStart()` methods will not run correctly, and you will see a warning similar to the following:

"Assets/…/myScript.cs warning CS0144: myScript.Start() hides inherited member"

! If your script is part of an object that has the dontDestroyOnLoad flag set, not calling base.Start will prevent re-initialization of your script when a scene is loaded.

In other words: don't worry unless you need above mentioned behavior. In this documentation, as well as in the demo scenes, we often do not change Start(), nor do we invoke base.Start() simply because we do not need the added capabilities.

cfxNotificationIntegratedReceiver Properties

All scripts that inherit from SIP have properties that you can set from script or via the Editor:

- Verbose:
  If this is checked, the script will output notification information to the debug console whenever a notification is received.

- Sleep On Deactivate:
  If the Object is deactivated, the script will also ignore notifications when this is checked. Please note the difference to the following option

- Sleep On Disable:
  If this is checked, the script will ignore notifications when the script (not the parent object) is disabled. This assumes that the script has subscribed to notifications in its lifetime before it got disabled.

- Filter Using:
  Here you can choose which method for notification filtering is chosen (for details on filtering, see later in this documentation).

Working with the Notification Manager

The key to using the notification manager is realizing that you are no longer tied to a limited number of `Update()` and similar events, but have complete freedom in defining events and how to shape them. SIP comes with an array of convenience functions, and it opens up the raw power of notifications, unfiltered und unfettered, to those who always boldly code where no scripter has coded before.

To use notifications, we only need to understand three simple terms, and how they apply to the notification manager:

"Subscribe" to notifications

To allow high-performance, the notification manager doesn't broadcast any notification to all objects – it only notifies those objects that have 'subscribed' to a notification, i.e. indicate that they want to be notified. When an object subscribes to a notification, it must tell the notification manager the name of the notification it wants to be notified of. That name can be any string – the notification manager doesn't care – but your script will be only contacted when a notification is sent out with an exact match for that name. There are no restrictions on the names you can use; there are no reserved words – but it is up to you to make sure that you use the names consistently.

!	Make sure that you find a naming system for your project that makes sense to you, and that you are unlikely to screw up. Because if there is one drawback that comes with this amount of flexibility it is that it's a giant PITA to debug. If you misspell a notification name, no compiler in the world can help you. It's not a bug. It won't crash your scene. But it will, silently and stubbornly, refuse to work until you found the spelling mistake.

To subscribe to a notification, all a script needs to do is to call subscribeTo with the name of the notification, e.g.

```
    subscribeTo("EnemySighted");
```

From now on, whenever any script in your scene sends out a notification named "EnemySighted", your script will get notified. A good place to subscribe to notifications is in the Start() method.

!	You should **never** subscribe to notifications during Awake(). This is because at that point in time the notification manager may not yet be awake itself.

Implement `OnNotification()`

Once your script has subscribed to a notification it does not need to actively wait for it. As soon as another (or even the same) script sends out a notification with the subscribed name, the notification manger will call your script's `OnNotification()`

method. Inside `OnNotification()` you then do whatever is appropriate once you know that something has happened.

So the next step to do whenever you subscribe to one or more notifications, is to implement `OnNotification()` in your script. If you don't your script won't crash (it already has implemented a backup `OnNotification`), but you won't get the notification.

```
public override void OnNotification(string theNotificationName)
{
	Debug.Log("Received: " + theNotificationName);
}
```

Note the **override** keyword. This signifies to you that there is already is an OnNotification method implemented (the backup we talked about earlier), and that you intentionally want to replace the backup with your new version. Should you ever omit the 'override' keyword, the compiler will complain (first indication), and your code won't be called (dead giveaway).

! `OnNotification(string theNotificationName)` is **only one of many** convenience implementations SIP provides – it also offers other variants that you can use, depending on your needs. Just make sure that you **only implement one** `OnNotification()` variant per script – unless you really know what you are doing (hint: you don't). Please see the reference section

STOP If you are using VisualStudio or MonoEdit to write your scripts, their smart Auto-Completion may suggest that inside of the new method you call the overridden base method (e.g. `base.OnNotification()`) – **don't ever do that**. This will always result in erroneous warnings, and can result in unpredictable behavior.

Should your script subscribe to multiple notifications, they all come in via the OnNotification method, so your first order of business would be to check which notification you received, and act accordingly.

Here's a tiny script that will output "Jump was pressed." whenever any script sends the 'jumpPressed' notification.

```
public class waitForAKey : cfxNotificationIntegratedReceiver {

    void Start () {
        subscribeTo ("jumpPressed");
    }

    public override void OnNotification (string notificationName){
        if (notificationName == "jumpPressed")
           Debug.Log("Jump was pressed.");
          }
     }
```

Note that the script is a cfxNotificationIntegratedReceiver, not a mere MonoBehaviour, so it comes with full notification capabilities. In the Start() method, we subscribe to be told about any ‘jumpPressed’ notifications.

More importantly, note that **this script has no Update() method**. It is only called during notification, making it much more efficient.

In OnNotification, the script simply writes to the debug console. We know that OnNotification can only be called for this one notification, so we don’t necessarily have to check if the notification received matches the one subscribed to. We have no illusions that you will follow above example. So, let us at least note that it is much better practice to always put a guard

```
      if (notificationName == "jumpPressed")
```

in front of your code, even if you are absolutely sure that this is not required. Oh, and see the section on SIP Best Practices for more annoying preaching.

Note also that we are using the most basic version of OnNotification(), there are more elaborate versions available to you that we’ll describe later (see the Reference section).

The counterpart is another script (or code inside the same script, but where’s the fun in that?) that detects if the “Jump” button is pressed, and then posts (or sends) a notification “jumpPressed”. Below example utilizes the cfxNotificationSender class, a simpler SIP class that can only send notifications, not receive them. We use it for purely educational purposes, you might as well have used the cfxNotificationIntegratedReceiver class:

```
public class detectjumpkey : cfxNotificationSender {

// Update is called once per frame
void Update () {
        if (Input.GetButton("Jump")) sendNotification("jumpPressed");
    }
}
```

(note that this script continues to send notification events as long as the ‘Jump’ button is pressed)

This simple mechanism is usually enough for 99% of all your requirements – but sometimes you need more flexibility, and the notification manager provides for this: Instead of using OnNotification(), you can tell the notification manager to invoke different methods for different notification names. As this is advanced stuff, please see 'Selective Notifications' in Geek Sector A, later.

Receiving notifications on inactive objects/disabled scripts

Before we go into details, we need to make an important distinction that is not obvious, and is a result of how scripts and GameObjects relate to each other.

In Unity, scripts are components of GameObjects, and each can be enabled or disabled individually, with slightly different semantics.

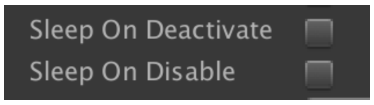
- A GameObject can be "activated" through the "SetActive()" Method. An object that is not active will not be rendered, and a script component will not receive Update() calls.
- A script inside a GameObject can be enabled or disabled by setting its 'enabled' property (e.g. theScript.enabled = true;). Disabled script's methods are usually not called by Unity

In this context we must therefore differentiate between four different states that can influence if a script's method is executed:

- Disabled/Enabled: the script itself is enabled or disabled
- Active/Inactive: the parent object is active or inactive, independent of the script's status.

SIP knows how to handle each of these, and gives you additional control to override Unity's default behavior:

The notification manager can send notification to objects and scripts that are inactive or disabled. Properties provided by the notification class allow you to control this behavior:



Sleep On Deactivate ☐
Sleep On Disable ☐

Each script has two properties that control how they react, and it's up to you to decide if the script should ignore notifications while inactive, or keep processing them. This was a deliberate decision to be able to tell an inactive object or disabled script via notification to become active/enabled, and to suppress this behavior when required.

- *Sleep On Deactivate*
  This controls how a script responds to a notification if the parent object is inactive. When told to sleep (checked), a script will not react to a notification if the parent object is not active.

- *Sleep On Disable*
  This controls how a script responds to a notification if the script itself is disabled (note that even though a script is disabled it can receive notifications if it subscribed prior to being disabled). When told to sleep (checked), a script will not respond to a notification when the script is disabled.

If a script is subscribed to a notification before it is disabled (the script, not the object), it will also continue to receive notification and act on them, unless you disable this ability.

Note that this feature is exclusive to SIP, and does not extend to other script methods.

Sending out Notifications

So, how do you send out a notification? Let's say, you are writing the AI script for a radar. What should it do once it detects an enemy? All it needs to do is send, or post, the notification, and all subscribers to that notification will be notified. At the most basic level, you would simply

```
sendNotification("EnemySighted");
```

and all scripts that have subscribed to "EnemySighted" will be notified. In real-world implementations, this could be an alarm klaxon, a starter gun going off in a race, a traffic light changing colors, or a doorbell ringing.

For historic reasons and convenience, SIP implements two whole sets of methods that accomplish the same: you can use either 'sendNotification' or 'postNotification' as method name, and interchange them whenever you like.

Sending notifications during Start()

It's usually not a good idea to send out a notification during your script's Start() method, because you can't be sure that the script that should receive the notification has already subscribed to it. If you absolutely must send out notifications during Start(), make sure to either schedule a delayed notification (see later in this documentation), or use the specially provided DelayedStart() method. In any event, please read the section on *When? Signing up for, and sending Notifications/Queries* (also see later).

Providing Context

It's often useful to provide some context with a notification to further aid smart development. In the examples above, we mentioned "a traffic light changing colors". This is a classic example of a notification that can increase usefulness if it not only notified us of the fact that the color changed, but also which color it changed to. SIP supports this out-of-the-box. You can pass a string with the notification to give your notification context. This is supported on both the sending and the receiving end:

```
sendNotification("TrafficLightChanged", "toGreen");
```

will send a notification named "TrafficLightChanged", and attach the string "toGreen" to the "Event" part of the notification (more on this later).

However, in order to be able to receive this additional information, you must slightly enhance your `OnNotification()` method:

```
public override void OnNotification(string
theNotificationName, string theEvent)
{
     Debug.Log("Received: " + theNotificationName + " with "
               + theEvent);

}
```

Note the second string variable theEvent that now contains the event. You can use this inside your `OnNotification`() to easily distinguish between the various possible events that have led to the notification. And yes, there are many more variants of OnNotification pre-defined for you in case you need even more information. Read about that in Geek Sector A, later.

Note that even though we call the second parameter "Event", you are by no means obligated to use it just for discerning events. You can use it any way to conveniently pass a string with the event. You could, for example define a notification named "PlayerWon" and pass the name of the player who won as Event. It really doesn't matter, as the notification manager will happily pass along any string you give it.

The reason we called it "Event" is because we built in another cool feature that you can, but don't have to use: event filtering.

!	You can pass **any** information with notifications by providing simple dictionary that contains any information you want. Please refer to Geek Sector A for more information

Controlling who's invited to the party: Blacklist vs Whitelist

For most Unity users, the combination of a notification name and a qualifying string that allows you to tell apart different notifications of the same name completely suffices.

But with comfort grows appetite. And with experience grows ability. Soon you'll start dividing your game's functionality by notification and events, and new requirement arises: the ability to filter events for your objects. This is because your objects now routinely subscribe to a host of notifications, but should only react to a subset of all events that a notification reports.

• • •	If your eyes just glazed over at the word 'filter', you can safely skip this chapter now, and come back when you discover the need to selectively pick events from a notification. It's more a convenience than a requirement to understand this chapter.

In short, wouldn't it be great if your script only reacts to some of the events you subscribed to, not all of them? For example, the InputReporter prefab that comes as a standard prefab with SIP reports events for both KeyDown and KeyUp.

Let's say that you are only interested in the KeyUp event.
The notification manager allows you to automatically "pre-approve" events for you, a process that we call 'filtering'.

! SIP comes with a full set of filters that all operate on the “Event” level of a notification. If you post events without setting the Event string (either using an appropriate call, or setting it manually), they won’t be filtered.

You have the choice of using two different ways to have events filtered for you, and you should choose the one that feels more natural: using a 'blacklist' or using a 'whitelist'.

Initially, you'll find that using a blacklist feels more natural (a blacklist defines a list of events that will not be let through the door), while after some experience you'll find that you'll be using whitelisting more often (a whitelist refuses entry to everyone unless they are on the whitelist). No approach is better than the other, they are merely convenience functions that make using the notification manager even more pleasant, so we encourage you to try both.

! When you filter notifications, it doesn't matter which OnNotification method you override – they all receive filtering. This allows you to implement some powerful tricks (with perhaps questionable wisdom) where your pre-filtered OnNotification method knows that it only receives notifications of a certain event.

Filtering events is easy. First, you need to choose the filtering method – black- or whitelisting – and then you simply list the notifications, and the events within each notification that you want to be placed on the list.

For example, you may be using the provided ReportInput prefab to centrally look at Input and broadcast Input events. One of your script is only interested in the KeyDown event that ReportInput broadcasts, not the others. It therefore puts “KeyDown” on the Whitelist:

```
public class keyDownOnly : cfxNotificationIntegratedReceiver {

    // Use this for initialization
    void Start () {
        subscribeTo ("Input");
        filterUsing = cfxFilterMethod.WhiteList; // use whitelist
        filterEventForNotification ("KeyDown", "Input");
       // only allow "KeyDown"
    }

    public override void OnNotification (string notificationName,
                                         string theEvent) {
        // should only receive "KeyDown" as theEvent
        Debug.Log ("Received event: " + theEvent);
    }
}
```

In Start() we

- Subscribe to the “Input” notification.
- We then set filtering to Whitelist via script instead of using Unity Editor. This means that only those subscribed notifications that match specific strings (that we’ll provide in the next line) in their “Event” field will be accepted.
- We add the event named “KeyDown” to the list of “Input” notifications that can pass (whitelisting). Note that when we use white- or blacklists, we always provide them in pairs: the event name and the notification name.

In OnNotification() we simply log all notifications that we are receiving. If the filter works correctly, we should only log KeyDown events, even though our script also receives KeyUp events from the “Input” notification

Scheduled / Delayed / Timed Notifications

SIP supports a powerful (and, once you get around to using it, convenient) feature: scheduled (or delayed) notifications. This comes into play whenever you want to implement a sequence of actions that are spread over time. Through notification scheduling you can achieve in a few lines of code what usually takes a lot more, perhaps even some kind of state handling. We'll see how simple this becomes directly after we show you how simple it is to schedule a single notification.

Background

On many occasions, it is helpful to think of notifications as reminders. You start a task, and then want to be reminded some time later to finish it, or do something unless the player has achieved a goal (the classic 'defuse the bomb in 30 seconds' scenario). Another common use is when you want to introduce a small delay and resume your actions after that delay (tutorials and cut scenes profit from this, as they usually create a sequence of timed events).

Scheduling a notification

You set up a notification to be sent out after a delay with this:

```
	sendNotification("OutOfTime", 30.0f);
```

This sends the notification named "OutOfTime" in 30 seconds. The delay can be added to all variants of sendNotification. If you set the delay to 0.01 seconds or smaller, the notification is sent immediately and is not scheduled.

Internally, SIP uses a co-routine for all scheduling purposes, so there is very little CPU overhead involved in using delayed notifications.

Unless you schedule a notification locally (see below), destroying an object that has scheduled pending notifications will not destroy the notifications.

Multiple scheduled notification

One of the more important shifts in your perception of how notifications can change the way you create scripts is to remember that you can schedule multiple notifications, which creates a sequence of events (actions). SIP has no upper limit on the number of concurrently scheduled notifications at any given time.

With SIP we have included a demo scene that simulates a common occurrence: at some point in time, you want to trigger a transient camera effect that

- turns on the camera effect
- fades the effect in over a certain time
- fades the effect out again
- shuts down the camera effect.

All this we can achieve with only three lines (Note: this assumes that the camera script knows how to handle the information. We'll discuss only the event handling code here, you'll have to study the shader code yourself if you are interested in how to do that. Be advised: it's totally worth your time to do so).

Here's the code that immediately starts the camera effect via notification, and then schedules two further notifications to fade back out, and then switch off the camera effect.

```
// 1. Provide context info, start fx fade-in immediately
Dictionary<string, object> info = new Dictionary<string, object> ();
info ["FadeIn"] = "fadein";
info ["Duration"] = "0.25";
sendNotification ("startFX", info);

// 2. Start fx fade-out, same duration, in 1.5 seconds
info.Remove ("FadeIn"); // configure for fade-out
sendNotification ("startFX", info, 1.5f);

// 3. Stop fx and turn it off 2.0 seconds from now
sendNotification ("stopFX", 2.0f);
```

Here are the relevant parts of the camera script that handle notification and turn itself on and off, and control the direction of the fade:

```
void Start() {
    subscribeTo (startCameraEffect); // they'll listen even
    subscribeTo (stopCameraEffect); // though disabled

    // disable this script
    this.enabled = false;
}


public override void OnNotification (string notificationName,
Dictionary<string, object> info) {

    if (notificationName == startCameraEffect) {
        // We use info to transport the following info:
        // time for the fx
        initialCounter = 1.0f;
        if (info.ContainsKey ("Duration")) {
            initialCounter = float.Parse (info ["Duration"] as string);
        }
        counter = initialCounter;

        fadeIn = false;
        if (info.ContainsKey ("FadeIn"))
            fadeIn = true;

        this.enabled = true; // turn on effect script (this)
    }

    if (notificationName == stopCameraEffect) {
        this.enabled = false; // turn off camera effect
    }
}
```

Note that this script takes advantage of the fact that it still receives notifications even though it is disabled.

SIP comes with a demo scene that contains the full source to this camera effect, plus the shader source for you to study.

You may also want to try your hand at implementing this sequence in code without notifications, and you'll see just how powerful scheduled notifications are.

Sending notification to "yourself"

A common design pattern is for scripts to send notifications to itself. You'll probably start doing that yourself very quickly. Doing so can result in the minor issue of coming up with notification names that are unique to this script and will not 'cross wires' with copies of the object in the same scene. SIP provides a convenient UniqueNotificationName() method for exactly this purpose. Please see the relevant chapter on how to use that powerful, yet simple-to-use method.

Local versus Central Scheduling

To account for Unity's special requirements we provide two different ways to schedule notifications. The difference between them is subtle, and in most cases SIP's default method (central scheduling) suffices.

When you schedule a notification, the underlying code sends the notification schedule to the central notification manager object which then schedules it via co-routines. This central approach is not only efficient; it also makes it possible that the notification is sent even though the originating object no longer exists – a common design pattern with scheduled notifications.

However, this can present you with a problem should you later want to cancel a scheduled notification. This is because the script that schedules the notification does not retain control about the notification schedule when it hands it off to the notification mananger.

Let's say you implement an "Alarm" notification that goes off unless the player manages to disarm the alarm before the timer runs out. If you scheduled this notification with the notification manger, your script can't cancel the scheduled notification should the player succeed in disarming it.
Of course, the notification handling code could check if the player has disarmed successfully and then ignore the notification; but it would be far better if your script could suppress the scheduled alarm notification in the first place.

Your script can retain control over its scheduled notifications by using the 'local' option for any scheduled notification.

```
sendNotification("Resolve", 1.0f, true);
```

The last parameter “true” tells SIP that you want to schedule the notification through the script locally. If you pass ‘false’, or omit this parameter, SIP schedules the notification via the notification manager.

The result of a locally scheduled notification is the same as using the central notification manager, yet has some preconditions:

- it requires that the object still exists at the time that the notification is scheduled to be sent. If you destroy the object before the timer runs out, no notification is sent.
- the script must be enabled when the notification is scheduled. Otherwise you will receive a runtime error - the Mono Framework isn’t amused if you try to schedule something on a disabled script). Remember that notifications can execute on disabled scripts if you allow it, and it is common to schedule notifications during OnNotification(). So bear this in mind when you schedule locally.

The advantage: If you schedule a notification using the local option, you can cancel any scheduled notification that was scheduled locally by that script with invoking

```
StopAllCoroutines();
```

Note that this will cancel **all** locally scheduled notifications. We at this point do not support selective cancelling of scheduled notifications, neither on the notification manager nor locally.

!	StopAllCoroutines() will do exactly what is says on the tin. Make sure that you are not running any co-routines of your own in the same script, or those will also be stopped.

When to use locally scheduled notifications

- when you may need to cancel the notification

When to use centrally scheduled notifications

- all other cases, especially when
    - the sending object will no longer exist at notification time
    - the sending script is disabled (but running anyway because it’s happening inside a notification)

A word on re-entrancy

“Re-entrant” is a scary word, especially with code, and for a few lines we will delve into geek territory – but don’t worry. All we are doing here is tell you that we’ve got you covered.

Here’s the thing: once you start recognizing how simple scripts get when you use notifications, you’ll enable more and more scripts with notifications, and naturally – perhaps inevitably – you’ll want to subscribe to a notification, or cancel a subscription during OnNotification.

This is natural because during a notification is when important things happen, and that is the most convenient moment to (un)subscribe or post a notification. From a coding perspective, though, this can pose a problem: adding a subscriber, or worse, removing one while at the same time the Notification Manager is busily running through the list of subscribers can introduce access conflicts.

The easiest way to avoid this is to simply disallow subscriptions or cancellations during notification time. But as we noted above, that is the most likely time you would want to rescind a subscription – or start subscribing to others: if your ‘guardian robot’ script received a ‘startup’ notification, it would be natural for that script to subscribe to ‘enemy sighted’ at that moment. If it receives a ‘powerdown’ notification, that would be the right time to unsubscribe from ‘enemy sighted’.

We therefore engineered the central notification routine to be fully re-entrant: you can send notifications within notifications, and six levels in, a subscriber can cancel any of their subscriptions without causing the world to end (but please read the caution note below).

So, relax and enjoy, for the answer to all the following questions

- can I send a notification within OnNotification()
- can I subscribe within OnNotification()
- can I unsubscribe within OnNotification()
- can I start Queries within OnNotification()
- can I start Queries in OnQuery()
- can I send notifications within OnQuery()

is “yes”.

So, yes, our notification manager is re-entrant.

There is currently one limitation: You should not add or remove responders to a query (see next section) while responding to a query in OnQuery.

!	If you remove or add a subscriber during a notification event, they will only be added or removed *after* the notification manager has completed notifying all subscribers from the currently active list of subscribers. This means that all new subscriptions or cancellations become effective only *after* the current notification event concludes and exits.

When? Signing up for, and sending Notifications/Queries

In Unity, there is no particular guarantee to the order in which scripts or objects are being awoken (read: Awake() invoked) nor their Start() method is invoked, except for the following:
Awake() is invoked before Start(), and before Start() is invoked, all active object's Awake() methods are invoked. Start() has been invoked before the first Update() invokes. This has some important consequences for the order in which you can use SIP:

1. **Never invoke subscribeTo() or respondTo() during Awake()**
   Awake() is the time when SIP itself is starting up, and there is no way to guarantee that it is already up and running by the time your script's Awake() is executing. If you are unlucky (and the world is built in a way that you will be unlucky at the worst possible time), your subscribeTo() invocation is processed before SIP is started, resulting in a lost subscription. At earliest, invoke subscribeTo() and respondTo() during your script's Start().

2. **Never invoke sendNotification() during Awake()**
   The same rationale holds as above holds true for sending notifications during Awake. SIP may not be up, the intended receiver may not have subscribed yet, and the notification will be lost.

3. **Never invoke sendNotification() or runQuery() during Start()**
   Similar to above, Start() is the time when all the scripts sign up for notifications. If you send a notification during Start(), you may run into a situation where the script that should receive the notification has not yet subscribed. At earliest, invoke sendNotification() or runQuery() during SIP's specially provided 'DelayedStart()", which guarantees that it invokes after all active objects have completed their Start() method.

! You *can* invoke sendNotification() during Start() if you schedule it with a delay of 0.01 seconds or longer. This is sufficient to guarantee that all other scripts have signed up.

If you do not adhere to above rules, you might run into strange intermittent phenomena like your scripts working perfectly in Unity, but not working at all when built as an executable.

! If you rely on DelayedStart(), remember that you must invoke base.Start(), and correctly prepend 'public override' to your void Start() message as follows:

```
public override void Start () {
    base.Start ();
    ... your code here
}
```

When? Execution time of Notifications

There's usually a lot going on in a Unity-based app, and to successfully synchronize your actions, it sometime becomes important to know just when (i.e. at which point in time during the render loop) notifications are run. So: when does OnNotification() or OnQuery() run (before, during, or after Update())?

Well, it's simple, but not that simple. As a general rule, OnNotification and OnQuery run during the same part of the cycle that a script invokes sendNotification() or runQuery(). This means that if you invoked sendNotification() (or runQuery()) during Update() – which will account for 99% of the time – the subscribed script will have their OnNotification() routine also invoked during that same Update(). If you invoked sendNotification() during fixedUpdate(), the notifications are sent out during that slice of the cycle. This is mostly because the notification manager runs on the same thread as the script that send the notification (or runs the query), and immediately executes the notification. In general (exceptions see below), it should be impossible for a notification to cross the boundaries of its time slice in the render loop: "what happens in Update() stays in Update()", so to speak.

Exceptions: delayed/scheduled notifications

Things are a bit different when you delay a notification via the built-in delay function. The notification manger uses a co-routine for the timing delay, and co-routines run after the Update() process completes, i.e. when you send a notification with a delay, all objects have already received their update() invocation. This holds true irrespective of centrally or locally scheduled notifications.

Note that this doesn't appliy to scheduled notifications with a delay smaller than 0.01 seconds, as they are not deferred, but send out immediately.

Notifications sent out by SIP Prefabs

- All timing prefabs in SIP send their notifications after Update() completes (they use co-routines for timing purposes)
- Audio Listener sends after Update() completes (they use co-routines)
- Animation Listener: after Update() completes (called during the animation cycle)
- OnCollision, OnTrigger : before Update() starts (during physics processing)
- Input Listener: During Update()

When in doubt, please look at the Script Lifecycle Flowchart contained in Unity's on-line manual.

Notifications received through the network

SIP provides you with the ability to send and receive notifications through the network. Since there are no provisions in Unity to synchronize two or more clients, all we can say that the notifications go out to the network at the times we described above; SIP will not delay or schedule outbound network notifications in any way.

At what point in time the network code feeds in an event received from the network is currently unclear; we are looking into the possibility of ensuring the notification broadcast only happening at certain pre-determined times.

Part II: Queries – Ask your Scripts!

So now that we have an easy means to shout out to other scripts, the next logical step up would be an easy way for the scripts to respond, similar to a 'roll-call' or vote. We have added this ability to SIP as a powerful means to instantly gather real-time data. While a system like the Query Manager makes only limited sense in classic monolithic apps, it works incredibly well in Unity's distributed script environment.

A key aspect of the Query Manager is the fact that your script doesn't know how many scripts will respond. The response therefore is a List of a varying number of objects. Although we have made the interface for using the query manager as comfortable as the notification manager (multiple formats, great flexibility, pre-defined dictionary entries), using the query manager is slightly more involved than the notification manager because you can return an answer.

Signing Up to be Queried

The mechanism is similar to the notification manager: you sign up to respond to queries, and when a query arrives, you look at what is asked, and respond accordingly. You can choose to return 'nothing', which means simply returning a 'null' value. To tell the query manager that your script is available to answer a specific question, you sign up using

```
respondToQueryNamed("queryname")
```

Should a script then run a query with that name, the QueryManager will invoke the OnQuery() method in your script.

In the rare event that you always want to be queried, no matter what the query is, you can either supply '*' as query name, which the Query Manager will interpret as 'Anything', or use the convenience method

```
respondToAllQueries()
```

!	This works only in one direction: while your script can choose to be queried every time, you cannot run a Query with a name of “*” that will query every script.

!	Even if you invoke respondToAllQueries, that script's OnQuery method will only be invoked on a scoped query if it is included in the scope. Please see the section on query scopes for details.

Responding to a query

When you use the convenience methods provided by SIP, the query is routed to one of the OnQuery() methods, of which you must implement exactly one per script. The most basic version is as follows.

```
public override object OnQuery(string queryName) {
     // do nothing, just return null
     return null;
}
```

Note the **override**, as SIP has pre-defined a host of smart OnQuery methods for you, and you must override one of them.

Unlike the notification manager, you can pass back a response to the script that has issued a query. As a return value your script supplies a generic object, which means that you can pass anything back, and it's up to you to ensure that the script that receives the reply knowns how to interpret the result. If you need to return more than a single value, the most common form is by using a Dictionary similar to the one that is used to transport the details of the query.

Due to the fact that most queries are usually of the 'who knows this' kind, we also provide a convenience form of OnQuery() that pre-fills the 'LookFor' parameter.

```
public override object OnQuery(string queryName, string lookFor) {
     // do nothing, just return null
     return null;
}
```

While its most obvious use is to provide a value that the script can compare against (e.g. a Tag), you can use it any way you like.

! There is one important precondition: if you assemble a query dictionary by hand, you must supply the "LookFor" keyword, or the OnQuery(queryname, lookfor) methods will not be invoked (because no LookFor keyword is present).

Running a query

Running a query is very much like sending a notification. There are multiple variants of the same method, and the most basic is

```
List<object> result = runQuery ("someQuery");
```

This will tell the Query Manager to send this query to all object that have signed up to answer the question named “someQuery”, and – and this is the big difference – compile a list of all their answers (if they return one).

Since most queries are usually of the form “look at <something> and respond only if it is <somevalue>”, SIP provides a convenience variant for this:

```
List<object> result = runQuery ("someQuery", "someValue");
```

Please refer to the Reference Section for the many convenience functions built into SIP. They are easy to remember once you see how they cascade, but needlessly muddy the waters at this point in the manual

Running a query at Start()

Running a query on Start() is usually not a good idea at the beginning of a scene - because this can introduce a race condition:

- Most objects sign up to answer queries do so during Start()
- The order in which objects have their Start() invoked at scene begin is not defined
- When your scrip's Start() method is invoked, it is by no means guaranteed that all objects in the scene that you intend to answer already have signed up. This means that the result from the query can be unpredictable.

If you really, absolutely must run a query at the very beginning of a scene, you have two options: use the specially SIP-provided "DelayedStart()" method (best practice), or use the following workaround in Start() to force it to run directly after the first Update():

- Subscribe to a 'startingUp' event
- Send a delayed notification for 'startingUp' and set the delay to 0.01 – this will schedule the notification (just barely) and send it after the very first Update() in your scene.
- Implement OnNotification, and invoke runQuery if the notificationName is 'startingUp'

Of course, if your object is instanced while the scene already exists, above does not apply. Please read the chapter on When? Signing up for, and sending Notifications/Queries for more important information on this subject.

Running a query with narrowed scope

Yes, we can see the question forming in your mind: what if I want to narrow down the search based upon the result of a previous query? What if, for example, I first gathered all my RTS vehicles, and now want to only get the tanks? Easy.

You can limit the group of objects that receive the query by supplying a list of responders – the so-called *scope* – with the query. If you have read the chapter on the notification manager's whitelist capability, this is the Query manager's version of that feature – "it's the same, but different".

If you pass a scope with your query, the Query Manager ensures that that the query is sent only to the objects in the scope list ("white list"), and additionally ensures that they have signed up for this query.

! When limiting a search with a scope, scripts that have signed up for all queries are only invoked if they are also included in the scope.

How to obtain a 'Scope'

But how do you get a scope? After all, one of the great features of SIP is that you don't need to know anything about the objects in the scene. Are we telling you now that all that was just marketing BS?

Certainly not. At least not that bit. SIP offers you many different convenience functions that conveniently blend out complexity of the many things you could do, but mostly don't want. And one of the capabilities of running a query is that SIP can return you a list of those objects that have responded with a non-empty result.

! Although all scripts that are eligible for a query are invoked, a scope list only contains the references to scripts that have responded with a non-empty result. A scope is therefore not a list of all the scripts that were asked, but all that have *answered*.

For convenience, SIP filters this information out unless you specifically request it. Once you have that information, you can use it as a scope for the next search.

```
List<string> allResponders = null; // scope returned here
List<object> allVehicles = null;
allVehicles = runQuery ("HasTag", "Vehicles", out allResponders);
```

In above example, allVehicles contains a List of all the responses OnQuery() returned. Additionally, allResponders contains a list of all scripts that answered to the query with a non-empty reply (important: the scope does not contain not links to the actual scripts, it contains internal designations that allow SIP to tell them apart). You can use the list of all responders to restrict a follow-up search to all responders simply by passing it with your next query.

```
List<object> allLand = null;
allLand = runQuery ("HasTag", "Land", allResponders);
```

This query is now restricted to those scripts that returned a non-null result to the initial query. Using the appropriate runQuery() variant, you can chain this multiple times, creating ever more precise queries (in effect, chaining them with a logical AND) using the results from a previous query:

```
List<object> allLand = null;
allLand = runQuery ("HasTag", "Land", allResponders,
out allResponders);
List<object> allTanks = null;
allTanks = runQuery ("HasTag", "Heavy", allResponders);
```

In above example, we re-used the scope variable allResponders to receive the responders of the second query, and used that to define the scope for the third query. Using the same variable as scope (in) and to receive the new scope (out) simultaneously *is* best practice, but may require some getting used to. SIP is specifically engineered to handle this case correctly.

Example: Multi-Tag

We will demonstrate the usefulness and flexibility of the Query Manager by implementing a feature that is woefully underdeveloped in Unity and that we hinted at above: a multi-tag system. As you probably know, you can ascribe a Tag to an object to indicate that it is part of a set. Unfortunately, in Unity's tag concept, an object can have exactly one Tag, out of a defined set of possible tags. We can take this concept, and make it more general: your objects can have multiple tags (i.e. belong to multiple sets). For example, in a typical RTS game, your units usually belong to multiple sets, i.e. have multiple tags:

- faction: Tags: red, green, blue
- domain: Tags: air, land, sea
  (Note: amphibious would have ‘land’ and ‘sea’ tags)
- class or size: Tags: light, heavy, super
- Upgrade Class: Tags: recruit, veteran, elite
- mobility: Tags: mobile, fixed
- exceptional Unit: Tags: standard, special

We can define a string that contains all tags for each object, and sign up to respond to "HasTag" queries. We use the LookFor form of OnQuery to respond with our own object only if the Tag that is looked for is contained in our Tags string.

Here's a possible Tag definition for an amphibious tank that belongs to the ‘red’ faction

```
public string tags = "red, land, sea, heavy, mobile, standard";
```

We now sign up to respond to the 'HasTag' query

```
void Start () {
        // allow inquiries to my tags
        respondToQueryNamed ("HasTag");
}
```

And finally, we implement the OnQuery response using the LookFor form:

```
public override object OnQuery (string queryName, string lookFor){
     // this will only be called if queryName matches "HasTag"
     // AND there is a "LookFor" string in info
     if (tags.Contains (lookFor)) {
           // yup, we have this tag, return game object
           return gameObject;
     } else {
           // null means nothing to see here, ignore my reply
           return null;
```

```
        }
}
```

A script that wants all objects with a 'land' tag would then run the following query:

```
List<object> landVehicles = runQuery ("HasTag", "land");
```

On return, landVehicles then contains a list of all currently active land vehicles.

Iterating the list of responders

As mentioned above, the responders to your query aren't anonymous, even though your script doesn't know them. Internally, the query manager always returns a list of who responded along with their responses; it's our convenience overlay that strips away this information, as it makes your code ungainly.

We now use the 'responder' variant of the method, to receive a list of scripts that returned something, that we then then use that list to narrow the selection further in the next iteration. So, let's first set up the query so we get a list of all scripts that returned something:

```
List<string> responders = null;
List<object> landVehicles = runQuery("HasTag", "land",
out responders);
```

Note the `out responders` part. This is where we'll receive the list of scripts that answered with a non-null result: the scope of the last query.

Using the scoped query invocation, we then further narrow the search down to all land vehicles that belong to the 'federation' faction

```
List<object> fedLandVehicles = runQuery("HasTag", "federation",
responders, out responders);
```

On the second runQuery call, we now used the result of the first query, limiting the search for 'federation' vehicles to the result of the first query, further narrowing the search (and in effect combining both searches with an AND). Using scopes, you can chain queries to quickly sort all your game objects.

There is a potential pitfall in above example: if you look closely, the final two arguments read `responders, out responders` - this is yet another variant of runQuery that not only takes an input scope (`responders`), but also returns a new scope (`out responders`). We used the same variable for both old and new scope as this is best practice if you want to quickly iterate through multiple tags.

We have included a slightly more elegant version of tags as the “Multi-Tag” prefab that also supports case insensitive search into SIP. You can add this prefab to your existing game objects and instantly upgrade them with this pseudo-tag ability.

Using DelayedStart()

SIP provides a special version of the Start() method that runs after all Start() messages of that frame have run (it executes after the first FixedUpdate(), but before the first Update() of your script).

Use this DelayedStart() to send notifications and/or runQueries that need to run at the start of your object's lifecycle, but that may run into a race condition with other objects that become active at the same time (usually at the beginning of the scene):

!

If you rely on DelayedStart(), remember that you must invoke base.Start(), and correctly prepend 'public override' to your void Start() message as follows:

```
    public override void Start () {
        base.Start ();
        ... your code here
    }
```

As we discussed with notifications, the notification manager is fully re-entrant. This means that you can post notifications, run queries, add subscribers and remove subscriptions

A note on re-entrancy

As we discussed with notifications, the notification manager is fully re-entrant. This means that you can post notifications, run queries, add subscribers and remove subscriptions during a notification event.

This is **not** true for the query manager. While you can run queries within queries without any problems, you **must not add, nor remove, responders while processing a query**. We might add this capability later, but currently view such behavior as bad design on your part.

A word on background processing

We have tested SIP with the notification manager running on a background thread. While not necessarily a smart option for notifications, running a query in the background is an interesting concept - especially if it involves collecting, compiling, and assembling textures, running queries on the Internet, or making sure that all level assets are loaded. While this works as long as you adhere to the Unity's rule to **never** access Unity's API on a background task, we found that this not only imposes too many restrictions to the experienced programmer, it's close to a gross negligence giving a lethal tool to non-experienced coders. It'd be akin to handing you a thermal lance when you expect a zippo.

We are now testing a different build of a background-enabled version of SIP that we may release at a future date - if we can make the risk more manageable, or at least provide some mechanism by which we can ensure that unsuspecting coders don't driver their processors headlong into a kernel panic. At least not unintentionally.

Using Queries as Notifications

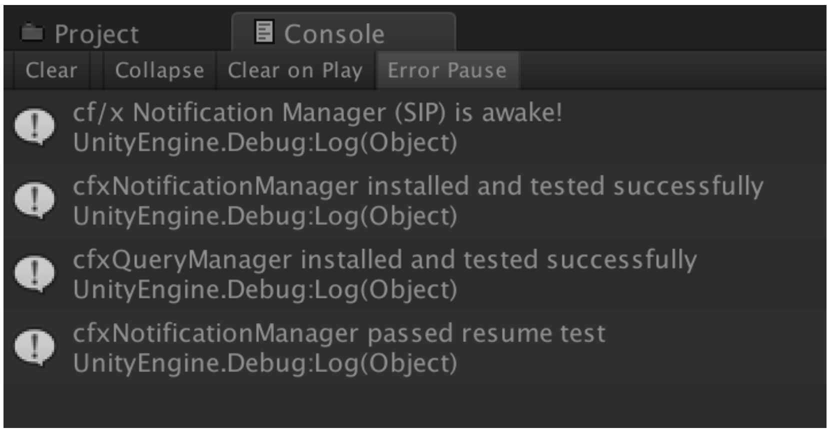
While the query manager builds upon the foundation of the notification manager, queries and notifications are not the same, and you should take care to tell them apart. The difference between asking a question and shouting out a message is obvious. That being said, there is a temptation to use the query manager as a form of ‘rhetorical query’; to use a query as an implicit form of notification. While it’s certainly possible to do this, we strongly advise against it. You should clearly separate these two concepts: queries are to gather information, and notifications spread it. Since queries are built to achieve a different purpose, they provide a different set of tools: they aren’t fully reentrant, can’t be scheduled, and do not provide filtering.

Part III: The Manager

SIP's heart is the tiny cfxNotificationManager prefab that must reside in each of your scenes. This little script manages all notifications and queries. You can remotely control this manager from any of your SIP enabled scripts as follows:

Self-Test

When SIP starts up, it performs a quick test of its main functions. Look for these lines in your log to make sure that SIP loaded correctly and is available to your scripts:



```
cf/x Notification Manager (SIP) is awake!
UnityEngine.Debug:Log(Object)
cfxNotificationManager installed and tested successfully
UnityEngine.Debug:Log(Object)
cfxQueryManager installed and tested successfully
UnityEngine.Debug:Log(Object)
cfxNotificationManager passed resume test
UnityEngine.Debug:Log(Object)
```

Suspending Notifications

Sometimes, for example when you need to pause the game, you may need to suspend notifications (note that if you implement pause by setting Time.timeScale to Zero it will usually not be necessary to suspend notifications).

You can at any time suspend notifications from being distributed by the notification manager by invoking

```
    suspendNotifications ();
```

on any of your scripts. From this point on, instead of sending out notifications, SIP is queuing all notifications for later sending. If you have scheduled notifications before calling suspendNotifications(), their timers will continue normally (accoding to Time.timeScale), but when they count down, they, too will be queued.

Resuming Notifications

SIP continues to queue all notifications until you resume notifications by invoking

```
    resumeNotifications ();
```

from any of your scripts. At that moment, SIP will execute all queued notifications in the order they were queued (including scheduled notifications that activated during suspension). If you scheduled a notification while the Manager was suspended, it will be scheduled now, with the delay unchanged.

Note that SIP allows you to discard all or specific notifications that are in the queue when resuming. Please see the reference section for more information.

Finding out if the Manager is suspended

At any time, you can invoke

```
    bool notificationsAreSuspended()
```

If this returns true, the Notification Manager is currently suspended.

! When suspended, usually Queries still work (i.e. return results) even though the utilize the underlying notification infrastructure. You can change that behaviour by setting an attribute. Please see below.

Accessing the Notification Queue

When you suspend notifications, you can access the notification names in the queue by invoking

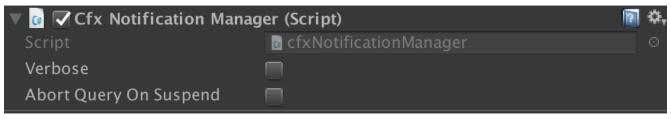
```
public List<string> pendingNotificationNames ()
```

this will return a list of all notification names that are currently pending and will be sent as soon as you resume the Manager. You can use this list to remove pending notifications before resuming notifications.

Suspended Notifications vs Aborting Queries

When you suspend notifications, the Query Manager continues to run normally. This is because while notifications function unilaterally (broadcast), Queries require a reply before they return a result. If the Query Manager also suspended queries, it would lock up the script. For this reason, the Query Manager and Notification Manager run on different virtual levels.

On rare occasions you may, however, want to suppress Queries from going out to other scripts when the Notification Manager is suspended.



Cfx Notification Manager (Script)

Script	cfxNotificationManager
Verbose	☐
Abort Query On Suspend	☐

To make all your queries abort while the Notification Manager is suspended, check the

```
abortQueryOnSuspend
```

property on the Notification Manager (or set it via script).

Part IV: Networking

SIP comes with astonishingly easy to use network support built in. This means that with SIP you can send notifications across the network as easily as you can broadcast them locally. With this you can, for example, build a network chat in just a few lines of code.

WARNING
The way SIP's network features work differs markedly from the way you normally implement networking in Unity. SIP does not require you to use [Server], [SyncVar] or [ClientRpc] attributes, nor will it require you to write different code for client and server.

Most importantly, though, sending and receiving network notifications is truly global, and unlike with Unity GameObject networking, sending a net notification is **not** restricted to objects with a network ID, nor is it restricted to objects the server knows about, nor will it cause the remote copy of the notifying object to issue the notification. When a net notification is sent out from any object, it is transmitted to the server via SIP-internal messaging, where it is then broadcast to all remote notification managers, which then broadcast the notification locally. If you are used to Unity's way of networking, you will probably be surprised by the simplicity of SIP's network broadcasts.

Receiving Notifications from the Network

When receiving networked notifications, SIP's network features are completely transparent. Unless you are explicitly looking for this information, your scripts won't even know if a notification was sent from a local script or the network; receiving networked notification requires no extra code. Any SIP-based script can receive networked notifications, and they do not differentiate between a network-received notification and a local one. Since SIP doesn't differentiate between networked and local notifications, to receive a networked notification, all you need to do is to subscribe to its notification name, just like with any other plain old boring notification. If a local or networked notification matches that name, your OnNotification (or other responder method you supplied) method will be invoked.

Sending Notifications across the Network

Usually, you don't want to send all your notifications across the network; this would be wasteful, and can lead to confusion with Unity's way of how it implements multiplayer support. So, most of your notifications remain local, and are sent with sendNotification().

To send a *networked* notification all you need to do is

- base your script on `cfxNotificationMPTransponder` (instead of `cfxNotificationIntegratedReceiver`)
- use `netSendNotification()` instead of `sendNotification()`

From this moment on, all your scripts can send and receive networked messages – with one small detail missing: they will not, in fact, be sent over the network. This requires another small item in your scene that we'll describe shortly. But it is important to know that you can start testing all your networked scripts locally without having any network-specific components in your scene.

To have networked notifications work across the network, you must

- have a network manager in your scene that descends from cfxSIPNetworkManager instead of NetworkManager (see below)
- be connected to another client via above mentioned cfxSIPNetworkManager

Note that to send networked notifications, your scripts do not have to be specially network enabled – although SIP does provide network-enabled (NetworkBehaviour- and NetworkManager-descendant) classes, if you need them.

Note also that all client/server management happens SIP-internally, so you do not need to check if you are connected to other clients or not. If you are currently not connected to a server of other clients, netSendNotification will have the same effect as sendNotification – it simply broadcasts a notification locally.

netSendNotification vs. sendNotification

Since netSendNotification builds on the normal sendNotification, it offers (almost) the same flexibility. You can't schedule netSend locally (but normal scheduling is supported), and you can't send complex objects as part of the notification, which is why there is no support for directly sending a GameObject with a notification (one of the overloads supplied with SIP for normal notifications). All other methods are supported, e.g.

```
netSendNotification ("NetTest", "Eventname", 1.4f);
```

Also, only the 'SendNotification' method tree of notifications is supported. For historical reasons, SIP also supports a 'PostNotification' method tree, which simply calls the same SendNotification method. We have done away with that inefficiency for networked notifications.

!	Before sending a notification across the net, SIP converts the info dictionary into a net safe format, stripping all entries that are not strings. Make sure you use SIP accessors to store non-strings in the info dictionary to transfer variable types like Float, Vector3, int or Color.

Restrictions placed on network notifications

SIP uses Unity's provided network infrastructure. This means

- You can use LAN and Internet-based networking out-of-the-box with SIP
- If you are using your own, or third party networking, you'll have to adapt the networking part yourself, which, depending on your requirements can range anywhere from trivial to challenging.

To maximize effectiveness, the network notification manager is itself a notification manager script, that contains only a few lines of true network-dependent code, and is controlled using – what else – notifications.

- Due to the way Unity implements networking, you cannot send complex objects across the network. This holds especially true for GameObjects. SIP enforces this by only allowing strings to be sent as payload with the info dictionary. If you use the SIP-provided addXXXtoInfo() and fetchXXX() accessor methods that encapsulate most data types, you can send all common data types (Float, int, Color, Vector3, bool, string). Please see the section on Accessors
- Currently, SIP only implements notifications across the network. All SIP Queries currently remain local.

!	Remember that any project that uses Unity's built-in network abilities must have its 'Multiplayer' settings enabled in Services before you can successfully connect two clients over the internet (LAN should work fine, though)

SIP places one restriction on networked notifications: only values in the info dictionary that is sent with the notification that are of string type are transferred over the network. This is usually not a problem, as SIP uses strings to transfer floats, ints and bools. It does pose a problem, however, when you use it to transfer GameObjects (including Prefabs). The 'Object' tag in the info dictionary is therefore never valid when you receive a networked notification.

The cfxSIPNetworkManager

Since SIP relies on Unity networking, for it to function in a networked environment, you must use the SIP-provided, creatively names cfxSIPNetworkManager class, which is nothing more than a notification-enabled NetworkManager class, that adds 20 lines of code to send out notifications to the network notification manager (that is part of every scene that has the cfxNotificationManager prefab).
If you extend the network manager yourself, you must base it off the SIP-provided Network Manager (cfxSIPNetworkManager) class, and ensure that you send the four required notifications at the appropriate time. SIP Network Manager is 100% backwards compatible with the normal Unity Network Manager class.

SIP is only able to send notifications to clients/servers that are connected through Unity's Network Manager (or cfxSIPNetworkManager) protocols.

The cfxNetworkedNotificationManager

This little gem, itself a non-networked cfxNotificationIntegratedReceiver script, is the unsung, and never-seen hero of all cross-network communication. If you are using the cfxNotificationManager prefab, it is already part of your scene, and in a few minutes, you'll probably have already forgotten about it, since there is nothing important you need to remember about it except that it exists. The networked notification mananger, sitting on top of the 'normal' notification manager does all network translation, up/downcasing notifications, and handles all server- and client states. If you want to look at a beautiful way to use SIP, look at that code.

How to differentiate between networked and non-networked notifications

From a scripter's perspective, there is next to no difference between a notification that was received via the network, or one that was initiated locally. There is, however, a sure tell-tale that you can use to tell the two apart: a notification that was sent across the network always has an entry in the info dictionary with a key = cfxNetMessageTag and a non-null value: the string “TRUE” so you can use the fetchBool accessor like this:

```
bool isFromNet = fetchBool(info, cfxSIPConstants.cfxNetMessageTag, 0);
```

! This only applies to notifications that were sent across the network – if your app is the one that issued the netSendNotification(), this instance will receive the local info dictionary without that key-value pair.

Setting up a project for networked notification

In order to use networked notifications, you do not need any special provisions. Enable Multiplayer in the Services tab of your project, then simply add the standard cfxNotificationManager object to your scene, and you are set to use all of the functionality, local and networked, that SIP provides. To be able to use netSendNotification(), make your script inherit from cfxNotificationMPTransponder instead of cfxNotificationIntegratedReceiver.

If you don't have a SIPNetworkManager in your scene, none of the notifications will be broadcast across the net, but all functionality works locally, without generating any errors. This was designed intentionally to enable you to use a common code base for networked and non-networked scenes.
In other words: you do not need to update a single line of code to switch between online and offline functionality). If no SIP Network Manager is present, netSendNotification and sendNotification behave the same: the notification is broadcast locally.

If you want networked notifications to be broadcast you add a SIPNetworkManager (or a script that inherits from cfxSIPNetworkManager) to the scene. Adding the SIPNetworkManager provides all the groundwork for being able to send and receive notifications across the network; however, notifications will only come in or can be broadcast to other apps once they are connected to each other through the NetworkManager. Once your apps are connected, netSendNotification will cause that notification to be broadcast to all connected clients.

Below small example broadcasts a notification with ‘Can you read this?’ as payload in the freely defined ‘message’ key to all connected clients (including the broadcaster) when you press the ‘space’ key on one of the connected clients:

```
public class networkTrial : cfxNotificationMPTransponder {

// Use this for initialization
public override void Start () {
    base.Start ();
    subscribeTo ("NetTest"); // Just like local notifications
}

// Broadcast message across net on space key pressed
void Update () {
    if (Input.GetKeyDown ("space")) {
        Dictionary<string, object> info = getBasicInfo ("Ignore");
        info ["message"] = "Can you read this?";
        netSendNotification ("NetTest", info); // net-wide broadcast
    }
}

void dumpDict(Dictionary<string, object> info){
    foreach (string key in info.Keys) {
        if (info [key] is string) {
            Debug.Log ("Info[" + key + "] = " + info [key]);
        } else {
            Debug.Log ("Info[" + key + "] = (not a string)");
        }
    }
}


public override void OnNotification (string notificationName,
Dictionary<string, object> info) {
    Debug.Log ("received notification: " + notificationName);
    Debug.Log ("Message is: " + fetchString(info, "message");

    }
}
```

If you run this small program in multiple instances on the same machine, or on multiple machines –and use Unity's NetworkManager HUD to connect the apps, you'll see they receive the notification whenever you press space. Yes, this is already 90% of a simple network chat app.

NetworkBehaviour, NetworkManager and NetworkLobbyManager

Usually, your scripts inherit from cfxNotificationIntegratedReceiver or (when you need network broadcasts) cfxNotificationMPTransponder. These classes inherit from MonoBehaviour, which gives you access to all the functionality you are accustomed to with regards to GameObjects.

However, there are three important classes that exist in Unity that are targeted for Multi-Player gaming and networking: NetworkBehaviour (which is similar to MonoBehaviour but requires your GameObject to have a network ID), NetworkManager (which handles Unity networking) and NetworkLobbyManager (a subclass of NetworkManager, which contains convenience functions to corral players and synchronize game start). SIP implements the complete notification methods for these classes, and allows your scripts to freely exchange notifications amongst each other, regardless of their class. So, if your script calls for a NetworkBehaviour script, use the cfxNetworkedNotificationIntegratedReceiver or cfxNetworkedMPTransponder classes, and your scripts gain full access to all SIP features. The same is true if you need to extend Unity’s NetworkManager or NetworkLobbyManager classes. Simply use the cfxSIPNetworkManagerIntegratedReceiver or cfxSIPNetworkManagerMPTransponder classes to gain full access to all SIP functionality for the NetworkManager, or the relevant versions for the NetworkLobbyManager.

As a good demonstration of how powerful SIP is, we encourage you to inspect SIP’s cfxNetworkManager it’s a cfxSIPNetworkManagerIntegratedReceiver class with only a few lines of code to send notifications.

!	SIP allows you to freely mix notifications from all SIP classes. A notification from a NetworkBehaviour is received by a MonoBehaviour-, NetworkLobbyManager- or NetworkManager-derived class without any additional coding.

Part V: Additional Features

Persistent Objects and SIP

Unity uses a special structure called ‘Scenes’ that logically group the GameObjects you create in UnityEditor. Each scene can be interpreted as a little self-contained app, and when you switch a scene (or level), Unity loads the new scene, while forgetting everything about the old one.
However, when changing scenes, most games want to remember some details about the last scene, for example how many lives remain for the player. There are many techniques for remembering such information, and Unity provides some. On method is that you tell an object that it can persist through multiple scenes: at the end of one scene such an object is not discarded, but it remains and lives in the new scene. You can easily accomplish this by using Unity’s DontDestroyOnLoad method on a GameObject. Such an object is no longer destroyed when a new scene loads. Since it’s already loaded, such an object also won’t have its Awake() nor Start() method invoked in the new scene.

This can lead to several potential issues with SIP-derived objects:

- If your script has subscribed to notifications, the scene’s new notification manager doesn’t know about your script – you have to re-connect to the new notification manager, and then re-subscribe to all notifications. You usually do this during Start(), but a persistent object’s Start() method isn’t invoked again.
- Furthermore, if your scene depends on your script needs to send out a number of notifications at the end of the Start() cycle, the same issue applies: SIP’s DelayedStart is only invoked once – and that was at least one scene ago.
- If you make the cfxNotificationManager a persistent object, it may remember objects from the last scene that are no longer around but that may have subscribed to notifications that are also sent in the new scene (a common occurrence if you are smart and re-use your objects). Also, a normal SIP-enhanced scene usually has anywhere from one- to two hundred subscriptions running. Carrying all this over to subsequent scenes will impact performance

To resolve these issues, SIP provides built-in functionality that you can use to enhance your own persistent scripts:

Automatic Scene Change Detection

All SIP scripts automatically detect if a new scene was loaded, and as a result immediately reconnect to the scene’s resident notification manager, requesting a new UUID (SIP’s internal tracking ID) and forgetting all previous subscription in the process.

!	If you override SIPs Start() method, you must invoke base.Start() for this ability to activate. If you don’t invoke base.Start(), that scrip can’t detect a scene change, and NewSceneStart nor NewSceneDelayedStart won’t be invoked.

}

Automatic Scene Change Detection and Networking

SIP implements scene change detection for all classes, as this behavior is especially important for NetworkManagers, which usually persist through multiple scenes of network play. If you use the SIP-provided classes, you have scene change detection built in.

NewSceneStart() and NewSceneDelayedStart

SIP implements two methods that you can override to be called at the beginning of a new scene. Override NewSceneStart() in your script to re-issue your subscriptions, and override NewSceneDelayedStart() to issue notifications or queries directly after all other the scene’s Start() invocations have completed.

! When NewSceneStart() or NewSceneDelayedStart() are called, SIP has already reset the SIP object: it has connected to the new NotificationManager, received a new uuid, and forgotten all previous subscriptions

You should separate out the notifications and subscriptions you issue at the beginning of a new scene, and arrive at the following pattern:

```
public override void Start () {
    base.Start (); // MUST be called!!!
    doSubscriptions();
}

public override void DelayedStart(){
    doNotifications ();
}

public override void NewSceneStart() {
    // we are persistent, and a new scene was loaded
    doSubscriptions();
}

public override void NewSceneDelayedStart(){
    sendNotification ("Ready");
}

void doSubscriptions() {
    subscribeTo ("NetTest");
}

void doNotifications() {
    sendNotification ("Ready");
}
```

!	If you override SIPs Start() method, you **must** invoke base.Start() or that script can’t detect a scene change, and neither NewSceneStart nor NewSceneDelayedStart will be invoked.

Never make the Notification Manager persistent

Making the notification manager a persistent object is a bad idea. It will hate you for doing so. Just don’t. OK?

Notification Accessors and Convenience

One of the powerful features of SIP is that you can send supplemental information with any notification. All information that you pass along with a notification is stored in a Dictionary as key-value pairs in the form <string, object>. Due to the importance of being able to transport this information, and some of the restrictions placed upon this dictionary under certain circumstances (e.g. transporting it across the network), SIP provides you with a host of accessor functions that not only follow best practice programming (resilience), but also ensures that the values stored are optimally compatible with SIP. If you use the accessor methods to store and retrieve values in the info dictionary, you can be sure that they always work correctly across multiple versions and networks – and you can utilize the same convenience methods we built into SIP's notification mechanism.

The 'implicit info' dictionary

If you override OnNotification in your script, you have the choice of multiple variants, not all of which provide access to the info dictionary that holds all supplemental notification information. However, SIP still holds that information ready for you – either directly in form of a variable you can access, or through "short-form" accessors that access this variable for you.

!	The implicit info dictionary is only set up for SIP's convenience OnNotification method tree. It is not initialized when you supply your own callback with subscribeTo().

UniqueNotificationName and SIP-uuid

A very common use of SIP is to use it in a script to send itself (usually timed) notifications. However, since notifications are broadcast, this can quickly lead to confusion unless you police the notification names – especially if you are re-using code as every good programmer does. And doubly so if you are creating objects from prefabs that should send itself notifications – but not others.

If for example, your AI script has a timer that fires regularly to tell your game object to choose a new action, you would want a way to be able to create a unique notification name that only this object knows about, and that will not cause other objects to react as well.

SIP provides a simple mechanism for this: the UniqueNotificationName() method that is guaranteed to create a new string based on the input string that is

- Guaranteed to be different from any other UniqueNotificationName() invocation from any other script in the same scene
- Will always result in the same string in the same script invocation

For example, UniqueNotificationName("ABC") results in "ABCcfxSIP-120-26" for every invocation in one instance of a script, while it results in "ABCcfxSIP-120-119" for each invocation from another instance of the same script in the same scene.

This means that you can use UniqueNotificationName() in a script to subscribe to a base notification name (“ABC” in above example), and each instance of that script in your scene will only receive the notification of it’s own invocation. Here’s a more elaborate example that subscribes to its own variant of the ‘newAction’ notification and then receives only its own notification every 20 seconds:

```
public override void Start () {
    base.Start ();
    subscribeTo (UniqueNotificationName ("newAction"));
    postNotification (UniqueNotificationName ("newAction"), 20.0f);
}

public override void OnNotification (string notificationName) {
    if (notificationName == UniqueNotificationName ("newAction")) {
        // time to pick a new action
        // ...
        // finally, remind myself to pick a new action in 20 seconds
        postNotification (UniqueNotificationName ("newAction"), 20.0f);
        return;
    }
}
```

You can use above script in an enemy AI prefab, and then populate your scene with lots of copies of that script. Each script will only receive their own notification, because each instance of that script has their own notification name based on “newAction”

So, how does SIP do this? The implementation is quite simple. Internally, all SIP scripts receive a unique ID (the script’s “uuid” attribute) when they contact the notification manager in a scene for the first time. This allows the notification manager to identify individual scripts and keep separate instances of the same script apart.

The UniqueNotificationName() method simply appends the script’s uuid attribute to the string you pass in. This makes the base notification name unique across the scence, yet repeatable inside your own script so you can reliably be notified of your own notifications withut triggering other instances of the same script in the same scene.

Note that this feature works reliably for persistent objects, provided you re-subscribe after scene load as described above (SIP internally ensures that after a scene load, the script re-connects to the new notification manager and retrieves a new uuid).

Retrieving information from the info dictionary

SIP supports a number of accessors for the most important types in Unity, allowing safe access and convenience at the same time. SIP currently supports accessors for the following C# and Unity types out of the box

- String
- Float
- Int
- Bool

- Color
- GameObject (only local)
- Vector3

If you use SIP accessors, with the exception of GameObject (which can't be transferred across the network) these types are also converted into network safe form, ensuring compatibility with netSendNotification().

All accessor functions come in two forms: the 'short form' that accesses the implicit info dictionary (and is only valid during OnNotification()) or the 'standard form', where you supply your own dictionary.

All retrieval functions allow you to optionally provide a default value in case an access fails. If you don't supply your own default value, SIP will supply it's own. The retrieval functions have the following format:

<type> fetch<type>([optional: dictionary,] key [, optional: default value)

For example, to retrieve a Vector3, this translates into the following versions:

- *Vector3 v = fetchVector3("theVector");*
  Access implicit info, use "theVector" as key, use SIP's default if an error occurs (SIP supplies [0, 0, 0] as default vector upon error)

- *Vector3 v = fetchVector3("theVector", Vector3.zero);*
  Access implicit info, use "theVector" as key, use the provided default if an error occurs

- *Vector3 v = fetchVector3(myInfoDict, "theVector");*
  Access myInfoDict, use "theVector" as key, use SIP's default if an error occurs

- *Vector3 v = fetchVector3(myInfodict, "theVector", Vector3.zero);*
  Access myInfodict, use "theVector" as key, use the supplied default if an error occurs.

Storing information into the info dictionary

There is no implicit info dictionary, so the convenience functions for storing information are simpler. The types mirror the retrieval methods:

Add<type>ToInfo(dictionary, key, value)

So, for example, to store a Vector3 in myDict, you'd write

```
addVector3ToInfo (myDict, "theVector", theDirection);
```

!	You might be tempted to forego addXXXToInfo, and simply go with `myDict[key] = value;` This will circumvent all SIP type conversion, won't work with some types, and is generally a bad idea, as it leads to unpredictable results. You may be smart, but we sweated the details – so it's better to trust us on this one ☺

Inspecting the info dictionary

SIP comes a convenience method to dump the contents of a Dictionary<string, object> to the log, so you can inspect it. To do that, simply call

```
public void dumpSIPDict(Dictionary<string, object> info)
```

SIP Sister Classes

SIP has three classes that achieve the same, but inherit from different base classes.

MonoBehavior	NetworkBehavior	NetworkManager	Description
`cfxNotificationBehaviour`	`cfxNetworkedNotificationBehaviour`	`cfxSIPNetworkManagerBehaviour`	Base class that implements all accessors, UUID, connection to the notification mananger, suspend & resume notifications, Start(), DelayedStart(), NewSceneStart, NewSceneDelayedStart()
`cfxNotificationSender`	`cfxNetworkedNotificationSender`	`cfxSIPNetworkManagerSender`	sendNotification() in all variants
`cfxIntegratedQuery`	`cfxNetworkedIntegratedQuery`	`cfxSIPNetworkManagerIntegratedQuery`	respondToQuery, withdrawQueryResponse, OnQuery, runQuery, runQueryAndFetchFirst, runQueryAndFetchRandom
`cfxNotificationIntegratedReceiver`	`cfxNetworkedNotificationIntegratedReceiver`	`cfxSIPNetworkManagerIntegratedReceiver`	subscribeTo, filtering, unsubscribeFrom, OnNotification
`cfxNotificationMPTransponder`	`cfxNetworkedNotificationMPTransponder`	`cfxSIPNetworkManagerMPTransponder`	netSendNotification
`cfxNotificationReceiver`	`cfxNetworkedNotificationReceiver`	`cfxSIPNetworkManagerNotificationReceiver`	SIP class notification branch that only implements sendNotification on top of cfxNotificationBehaviour

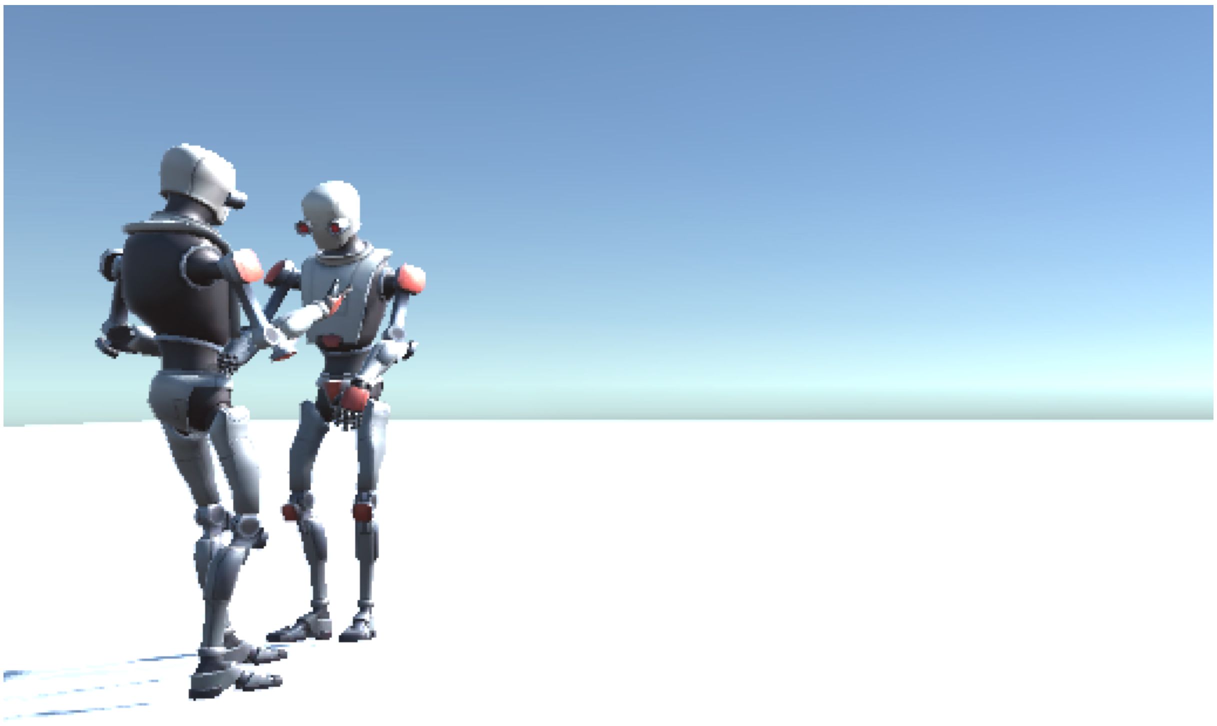
Part VI: Tutorial

This section is the SIP tutorial. We'll go through the provided demo scenes and explain what we did, and how we did it. The demo scenes all make heavy use of the SIP-provided prefabs, so we'll look into those as well. Each tutorial section has a corresponding Demo Scene in SIP, so you can look up the full code and scene. When in doubt, please also consult the reference section.

Scene01 – Synch Animation

Goals

This scene demonstrates how you can use SIP to easily accomplish something with SIP that is difficult otherwise: synchronize the actions of two actors (mecanim animations). The animation is simple: Once actor, 'Angry Kyle' is complaining to 'Bad Kyle', who is obviously bored. At one point, Bad takes a swing at Angry, punching him. Angry is hit, falls down and gets up again, while Bad laughs evilly. Then both animations loop.



The challenge

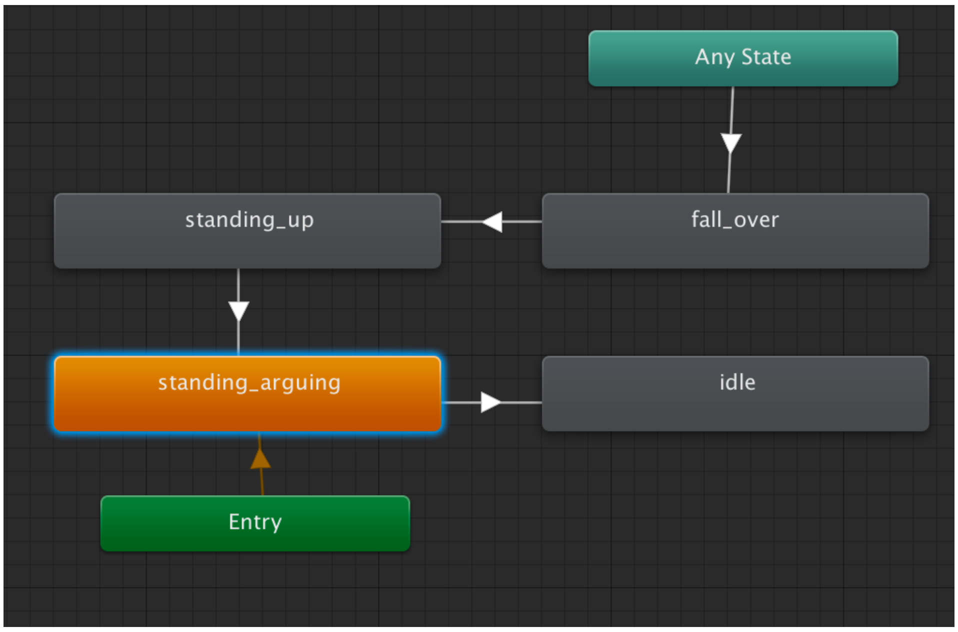
we have two actors that each have their own animations, and un-coordinated animation times. Once Angry is finished arguing, Bad is supposed to punch Angry, causing Angry to fall, and Bad will then laugh. The challenge in this is synchronize

Bad's punch (to coincide at the end of Angry's arguing animation) and Angry's reaction (hit animation, followed by falling down and getting up again). This is a common issue, and can be easily solved with SIP.

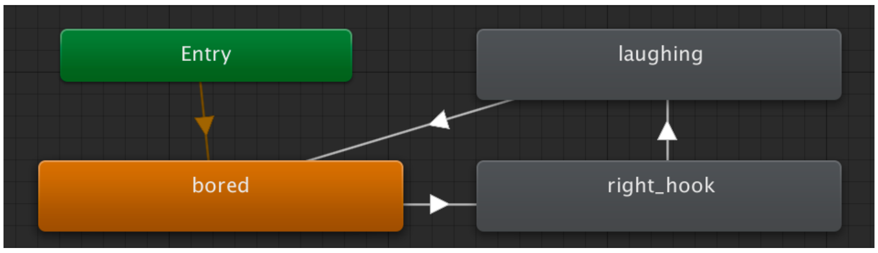
Implementation

Setting up the Animation Controllers

We first set up the two Animation Controllers for Angry and Bad. Angry plays the arguing animation and then transitions to idle. If at any time he receives the 'punched' event, he transitions to fall_over, then gets up, and then loops to arguing.



Bad plays the 'bored' animation looped. He transitions to right_hook on a punch event, which then flown into 'laughing' and loops back to 'bored'

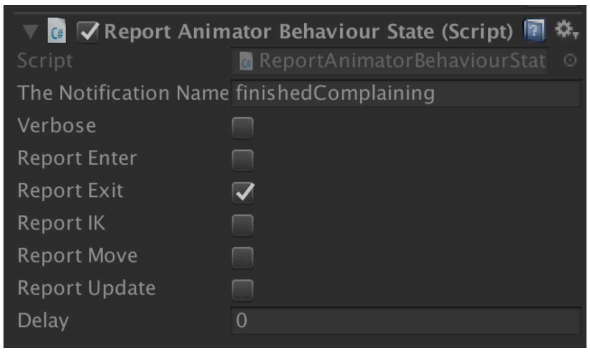


Synchronizing actions

We want Bad Kyle to hit Angry Kyle exactly when Angry finishes the 'arguing' animation. We are going to solve this common problem by using a SIP notification that is sent out when Angry's 'arguing' animation finishes. Bad will listen for this notification, and once received, First, we'll attach a script to Bad Kyle that listens for the 'finished' notification. In OnNotification(), which is only invoked when the notification is sent, Bad transitions to the punch animation.

So, the first step is to add the cfxNotificationManager to the scene. This enables all script-to-script communications.

Next, we'll add a ReportAnimatorBehaviourState, a SIP prefab written just for this purpose to Angry's 'arguing' animation. We set it up to report an exit, with delay of zero:



Report Animator Behaviour State (Script)

Script	ReportAnimatorBehaviourStat
The Notification Name	finishedComplaining
Verbose	☐
Report Enter	☐
Report Exit	✓
Report IK	☐
Report Move	☐
Report Update	☐
Delay	0

Next, we create Bad Kyle's script:
Create a new script component. First we change the class from MonoBehaviour to cfxNotificationIntegratedReceiver. Just changing this one word causes the script to automatically connect to the notification manager, allow it to send notifications, subscribing to notifications, and implements the OnNotification() methods of which we will override one. Attach this script to 'Bad Kyle'

```
public class punchAI : cfxNotificationIntegratedReceiver {

// Use this for initialization
void Start () {
    // the notifier script in the animator will send this event
    // when the model completes the 'Angry' animation
    subscribeTo ("finishedComplaining");
}

public override void OnNotification (string notificationName,
Dictionary<string, object> info) {
    // tell my animator to start the 'punch' animation
    Animator anim = gameObject.GetComponent<Animator> ();
    anim.SetTrigger ("punch");

    // send out notification that I just have thrown a punch
    sendNotification ("thrownPunch");
}

}
```

In Start() we subscribe to the "finishedComplaining" notification. This will cause the notification manager to invoke our OnNotification method any time any script in the scene send out a notification named 'finishedComplaining'.

Note that there is no Update(). This script requires almost no CPU.

To implement OnNotification() we must use the override keyword, so be sure to include it.

! If auto-completion is enabled in your script editor, it may now suggest that you invoke base.OnNotification. DON'T EVER DO THAT. This will cause SIP to spam you with erroneous error messages, reminding you that you need to override one of its OnNotification() methods. This is because when you invoke base, you just have looped back into SIP's logic that looks for an overridden OnNotification().

We know that there is only one actor that sends this message (Angry Kyle), so no further precautions are required (note: best practice would demand you guard this with an 'if' clause, but this is the real world). We grab the animator, and immediately send the 'punch' event, causing our Actor to start the 'punch' animation.

Now, if we had set up Angry's animation controller to immediately play the fall over' animation upon finishing the 'arguing' animation, this would be it. Both actor's actions are fully in synch, and we have achieved what we set out to do.

However, we want to make this a bit more interesting (if unnecessarily complex), and so we set up the animator slightly different: we queue an idle animation after 'arguing', and we transition to 'fall over' only on a 'punched' event. But this requires Angry Kyle to know when he was punched. In addition to starting his 'punch' animation, Bad Kyle therefore also sends out a notification of his own, to tell the world at large that he just punched someone:

```
sendNotification ("thrownPunch");
```

We'll now create a script for Angry Kyle that waits for that notification and then starts the 'receive punch' animation.

Create a new script, make it a cfxNotificationIntegratedReceiver, and attach it to Angry Kyle.

```
void Start () {
    // the other AI will send 'thrownPunch' when it starts the
    // punch animation
    subscribeTo ("thrownPunch");
}

public override void OnNotification (string notificationName) {
    // OK, the one is throwing a punch. Let
    // our Animator know so we initiated the punched animation
    Animator anim = gameObject.GetComponent<Animator> ();
    anim.SetTrigger ("punched");
}
```

In Start(), we subscribe to the "thrownPunch" notification that Bad will send.

In OnNotification we know that we are being invoked because Bad has punched us, and start the 'receive punch' animation. Now again both animations are in synch.

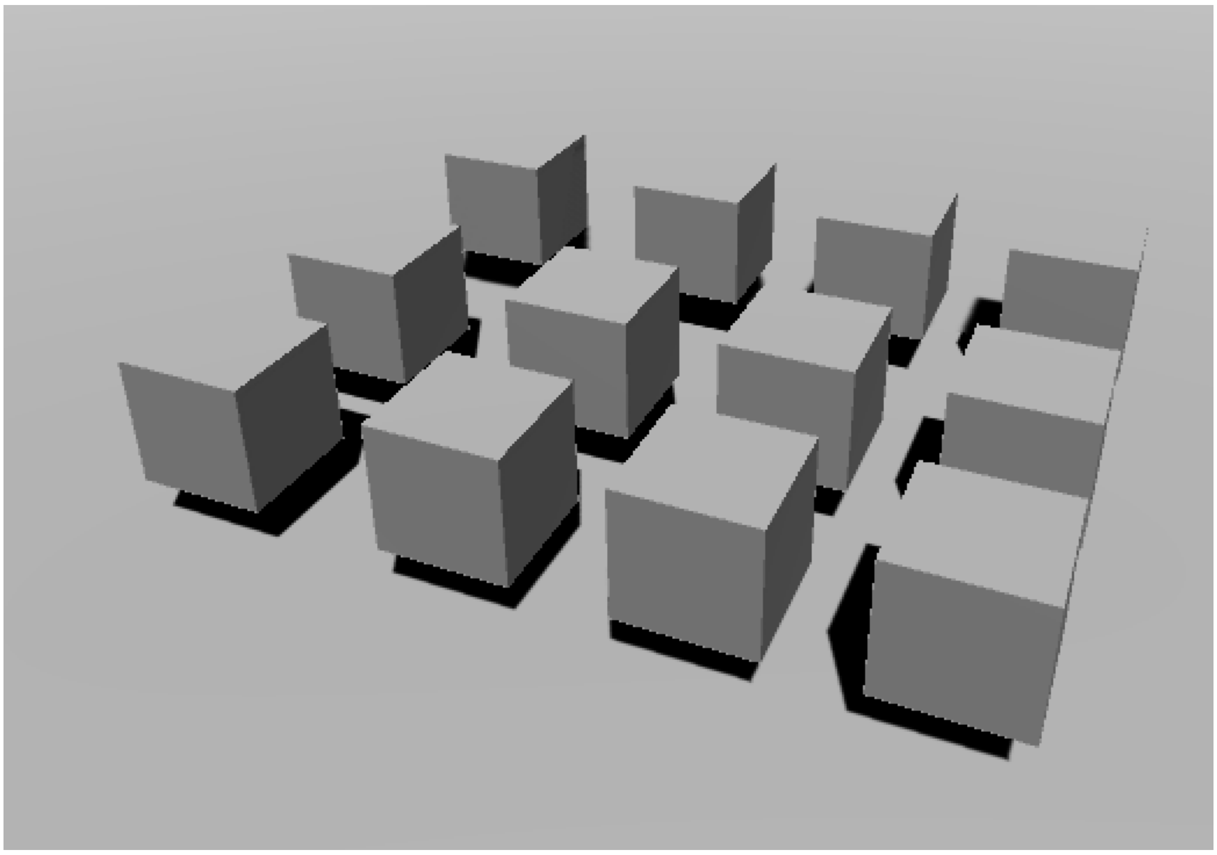
Observations

- It would have sufficed to merely synch Bad Kyle's punch; we did not really need the second script that we attached to Angry.
- What this demonstrates is that we can send notifications while processing notifications. When Angry receives the 'thrownPunch' notification, we are two levels deep into notifications:
    1. The Animation Controller sends the 'finishedComplaining' notification that is received by Bad.
    2. While processing that notification, Bad sends a notification 'punched' and this is received by Angry.
- Using SIP currently is the easiest general way to synchronize animations

Scene 02 – enable and disable scripts by notification

Goals:

Enable or disable scripts and objects via scripting



The Challenge

This is usually a tricky one – at least the enable part. The problem with enabling scripts is that in Unity, scripts that are disabled, or are parented by an inactive object, won't have their Update() invoked, making it difficult for them to enable or activate themselves. This usually means that they require an outside script that knows about the other scripts to activate them. Since SIP's OnNotification() methods can be invoked even when the parent object is inactive and the script itself is disabled, this becomes trivial with SIP.

Implementation

It couldn't get simpler. If you want you script to enable itself at some point in time, define a notification name accordingly, and subscribe to it. In OnNotification(), simply enable the script, and/or parenting GameObject.

First, drop the cfxNotificationManager into the scene.

Next create a cube and attach the following script:

```
public bool toggleObject = true;
public bool toggleScript = true;

// Use this for initialization
void Start () {
    subscribeTo ("toggleEnable");
}

public override void OnNotification (string notificationName) {
    if (toggleObject) gameObject.SetActive (!gameObject.activeSelf);
    if (toggleScript) this.enabled = (!this.enabled);

}
```

In Start() we subscribe to the event name that should activate/deactivate us – here we call it "toggleEnable"

In OnNotification() we re-enable our script and activate our GameObject according to the two public bools that we defined to enable or disable this ability.

In the demo scene we duplicated the cube a couple of times, and set the toggleObject and toggleScript bools to different values in order to test all, in conjunction with their SIP ability to sleep/wake on notifications.

We also use an input prefab to send the required notification. This is for illustration purposes only. We configure it so react to a button down ('jump', not key down), and make the notification name to match the name that the disabled script is listening for. The demo scene also uses code to toggle between states instead of simply activating as we illustrated above.

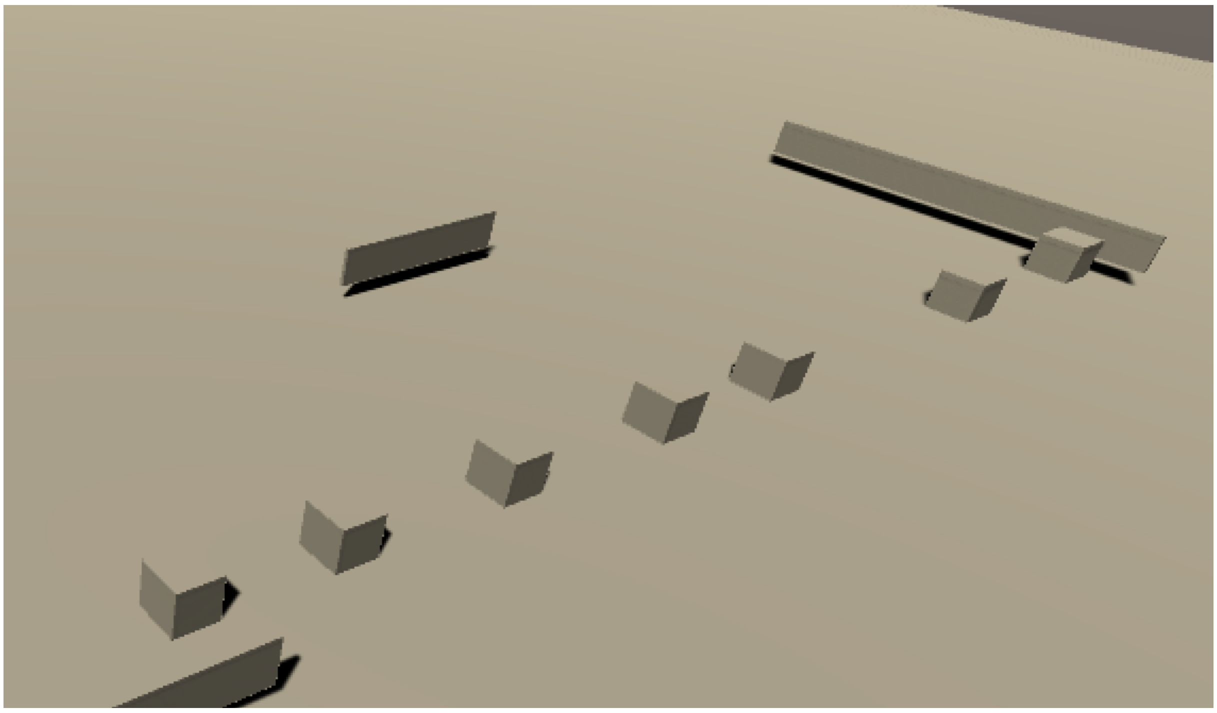
Observations

- Every cfxNotificationIntegratedReceiver script has properties that can control this behavior, and selectively disable either ability. Play around with the properties to understand how they work.
- The demo scene drives home the point of how elegant notifications are: we only send one notification, but can duplicate the receiving objects as often as we like, and all objects correctly enable and disable without needing to change a single line of code.
- The SIP enable/disable activate/deactivate prefab can add this ability to any asset

Scene 03 – collision broadcast

Goals:

A group of independent objects should react to a collision of one of their group with wall. This demonstrates a group of objects subscribing to the same notification to synchronize their actions, and respond as a group to a stimulus (also called ‘swarm behaviour’). It also demonstrates that multiple different objects can send the same notification.



Challenge:

The group should be truly arbitrary in size: the objects should not know each other and they should not link to each other, yet act in unison like a swarm.

Implementation

This is a classic example where notification managed systems like SIP shine. All the objects subscribe to the same notification. Although we could implement each object's OnNotification() separately (and therefore could have different behaviours for the same notification), we only implement the first script, add it to the object, and then cookie-cutter our swarm by duplicating the object multiple times and move the objects around.

```
public float speed;
private Vector3 lastPos;

void Start () {
    subscribeTo ("collisionDetected");
    lastPos = transform.position;
}
void Update () {
    lastPos = transform.position;
    transform.position += transform.forward * Time.deltaTime * speed;
}

public override void OnNotification (string notificationName){
    // turn by 180 degrees
    transform.position = lastPos; // last before impact
    transform.Rotate(0f, 90f, 0f);
}
```

Start()
We subscribe to the notification that one of us has collided. We also initiate lastPos, which we use for better collision handling (to avoid get stuck in a collider)

Update()
Each swarm object has an update method that moves it forward at a set speed and remembers the last position.

OnNotification()
When one of us collides with a wall, we turn by 90 degrees

Observations

- Multiple different objects (the walls) can send the same notification
- Nowhere in any script do we determine the size of the swarm. We could duplicate the entire swarm and without changing a single line of code in the entire scene it would continue to work.
- In the demo scene we took one object, and turned it by 90 degrees to make it head in a different direction
- Note that the game objects do not react when they collide with each other. Try to modify the script to change this – then do the same using the SIP collider prefab. Notice how simple it is to add collision broadcast to any object? Then watch in awe how complex suddenly your swarm's behavior becomes.
- We use SIP prefabs to make the walls report collisions. We thus added full SIP capability to a previously inert object. This is how you can integrate other assets just as easily.

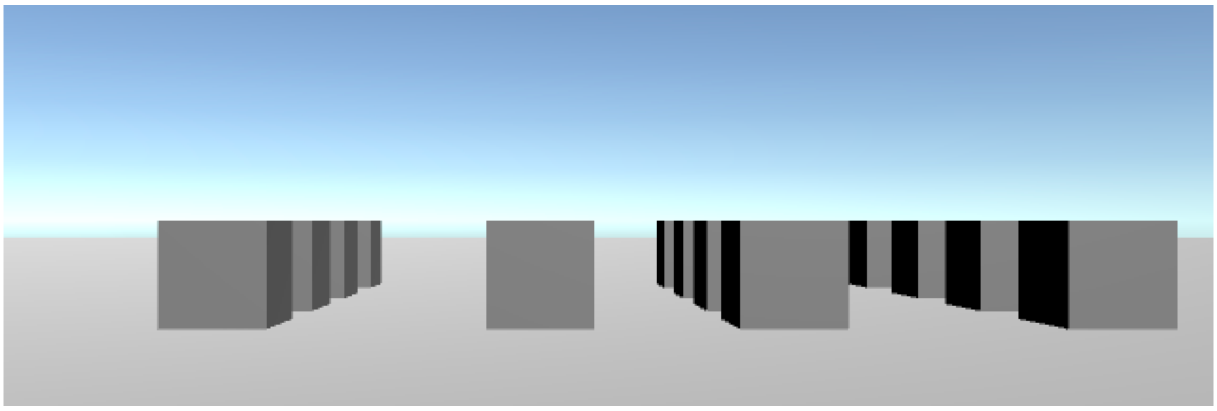
Scene 04 – simple stuff

[nothing much to describe here, really. We recommend you look at the scene and see if you can figure it out yourself.]

Scene 05 – basic query

Goals:

SIP's query system allows you to gather information from other scripts in the scene. We take the query system out for a spin to gather all objects of a certain kind.



Challenge:

You don't know anything about the objects that may respond: neither how many there are, nor what they are.

Implementation

All we want to find out in this first demonstration of query power is the game objects that can answer to my query.

Queries function similar to notifications, except that objects that receive a notification can respond. Each object that wants to respond to a query must tell SIP the questions that it is prepared to answer.

All objects that are to be gathered:

```
void Start () {
    // sign up for this query
    respondToQueryNamed ("RoleCall");
}

public override object OnQuery (string queryName){
    return gameObject;
}
```

Start()
We sign up to respond to a query “RoleCall”

OnQuery
Instead of OnNotification, you implement OnQuery. Unlike OnNotification, your method is expected to return a result or, if there is nothing to respond with, NULL.

Since this is a very simple demo, every script that signed up to respond answers with the parent gameObject

The object that runs the query

```
public List<GameObject> foundObjects; // will be filled by script
private bool hasRun = false;

void Update () {
    if (!hasRun) {
        // the rolecall query is implemented in all
       // cubes to respond with their game object root
        List<object> queryResponse = runQuery ("RoleCall");

        foreach (object anObject in queryResponse) {
            foundObjects.Add (anObject as GameObject);
        }

        hasRun = true;
    }
}
```

Start()
Note that we do not run the query here. There is no Start()!

Update()
We run the query once here. Note that it would be much better to do this outside of Update, usually as a result to a notification.
runQuery() receives an array as response that it must know how to interpret. In our case these are simply the GameObject links of all the objects that have responded. We iterate over all and add them to our foundObjects list

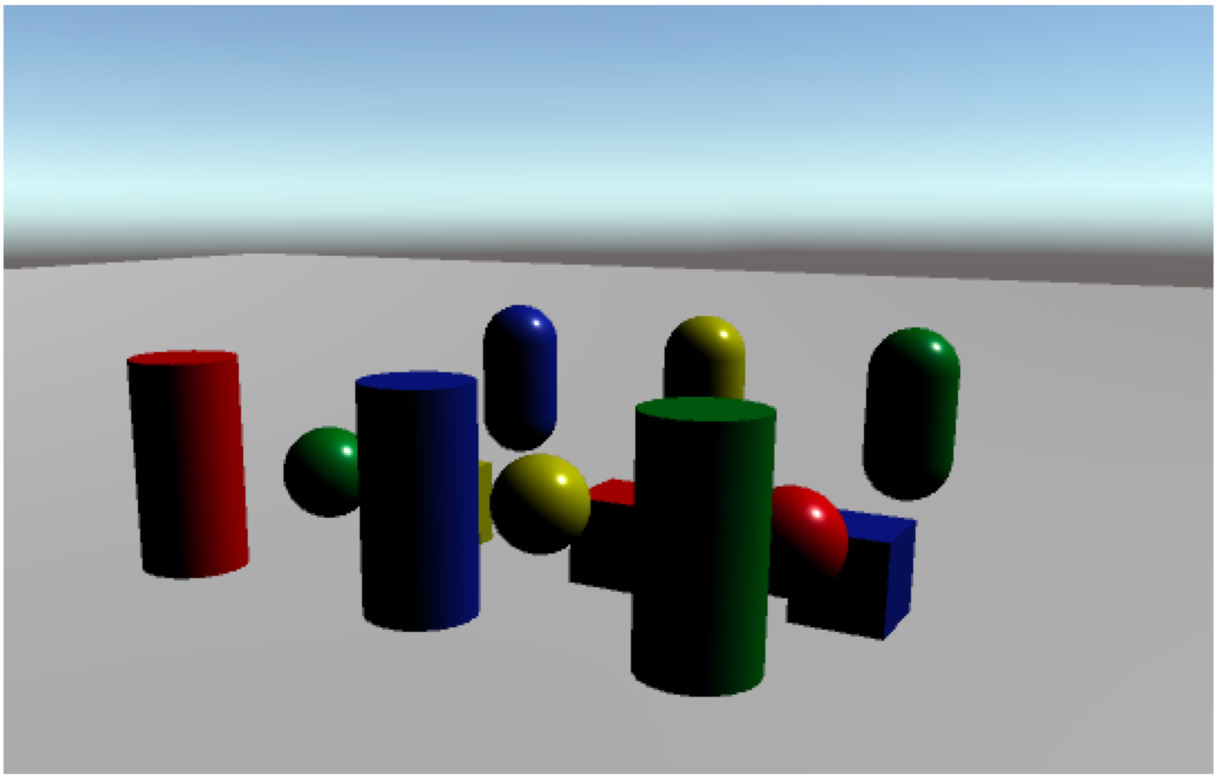
Observations

- Running a query during start is usually a bad idea, because at scene start, not all objects that sign up during start may have signed up yet.
- This almost trivial example is extremely useful, as it allows you to quickly gather an arbitrary number of GameObjects that your script then can iterate through.
- A good example for queries of this kind is when you gather object that may have been affected by an area of effect damage (e.g. an explosion). You could call the query "testforhit" and pass the location of the explosion and blast radius with the info dictionary. The scripts test if they are in range of the blast radius, and pass their game object if they are inside.

Scene 06 / Scene 07– better tags with queries

Goals:

Implement an in-game "Tags" system that can have more than one tags per object, additionally mix different implementations of OnQuery with the same query



Challenge:

Find all objects that have at least one specific tag

Implementation

All answering objects use a SIP tag Prefab that works as follows:
We contain all tags in one string, tags are separated by comma, e.g "Red, edge, Cube". This allows us to quickly find a tag with the 'Contains' method, but may be prone to false positives if the tags are chosen just slightly carelessly

```
public string tags = "Red, Cylinder";
public bool caseInsensitive;

void Start () {
    respondToQueryNamed ("hasTag");
}

public override object OnQuery (string queryName, string lookFor){
    string comparator = tags;
    if (caseInsensitive) {
        comparator = tags.ToUpper ();
        lookFor = lookFor.ToUpper ();
    }
    // this will only be called if queryName matches "Tags"
    // AND there is a "LookFor" string in info
    if (comparator.Contains (lookFor)) {
        // yup, we have this tag, return parent
        return gameObject;
    } else {
        return null;
    }
}
```

Start()
The object signs up to respond to "hasTag"

OnQuery()
We choose the variant LookFor for easy access to the string that we are looking for. All we need to do is to see if the string lookFor is contained in our tag string. However, to allow some leeway, we first convert both (tags and lookfor) to upper case to allow for a case insensitive search
If the lookFor tag is contained in out tag string, we return our GameObject, else NULL

The querying object

```
public void KickRedObjects() {
    string theTagWeAreLookingFor = "Red";
    List<object> objectsReceived =
            runQuery ("HasTag", theTagWeAreLookingFor);

    foreach (GameObject anObject in objectsReceived) {
        Rigidbody rb = anObject.GetComponent<Rigidbody> ();
        rb.AddForce (Vector3.up * 400f);
    }
}
```

KickRedObjects() runs a query with runQuery() that uses lookfor as convenient way to pass the tag string that we are looking for. The query manager returns a List of game objects that we can iterate (here, we give them a kick).

Observations

- Using a single string to store all tags will invite problems, as we use 'Contains()' to find a tag. A better variant is using List<string> for tags. We have supplied such a Prefab with SIP
- We use SIP prefabs in the demo
- The scene mixes different responders: One of the object uses the newer, better List<string>-based cfx Tag Prefab to demonstrate how easy it is to mix different implementation of the same query response.
- We implemented case insensitivity in this responder

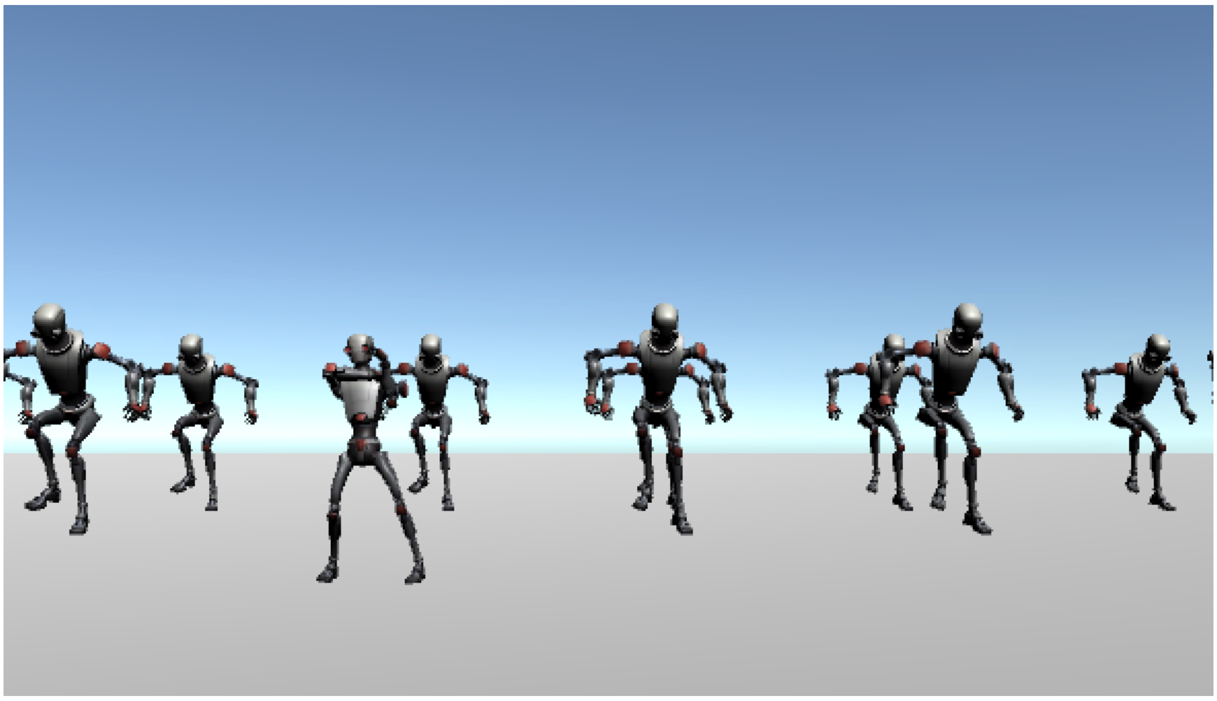
Scene 08 – Jump Ping

Goal:

Synchronize audio with animations

Challenge:

Unity has no good provisions to find out if an audio source as completed playing. This makes it difficult to synchronize actions to audio – be it music, or spoken words.



Implementation.

We use the AudioSource Prefab to be notified when the audio source has completed playing. The object that contains the audio source will also regularly speak the command "Jump", triggered by a timing Prefab. When the audio source finishes playing the audio clip, the Prefab sends out a notification

The Robot Actor has an animation controller for a looped dance routine, and a transition to jump when triggered by an event. It also subscribes to a notification to start the Jump animation.

Audio Source

```
void Start () {
    subscribeTo ("DoneSpeaking");
    subscribeTo ("Ping!");
}
```

```
public override void OnNotification (string notificationName){
    if (notificationName == "DoneSpeaking") {
            sendNotification ("Jump!"); // tell robots to jump
    }

    if (notificationName == "Ping!") {
        theSource.PlayOneShot (jump); // starts audio,
           // will eventually cause DoneSpeaking
         }
      }
```

Start()
Subscribes to the timing ping, and audio done.

OnNotification()
For the Ping notification we start the audio clip. This will eventually cause the audio clip to end, and the 'DoneSpeaking' notification to be sent

For the 'DoneSpeaking' notification we send out a new notification to tell the robots to jump. Note that we could have made it simpler if we simply let the robots subscribe to 'DoneSpeaking' but we want to demonstrate notifications.

Actor (Robot)

```
void Start () {
    subscribeTo ("Jump!"););
}

public override void OnNotification (string notificationName) {
    theAnimator.SetTrigger ("jump");
}
```

Start()
Subscribe to "Jump!" which is broadcast when the audio clip finishes.

OnNotification()
Trigger transition to jump animation on the controller.

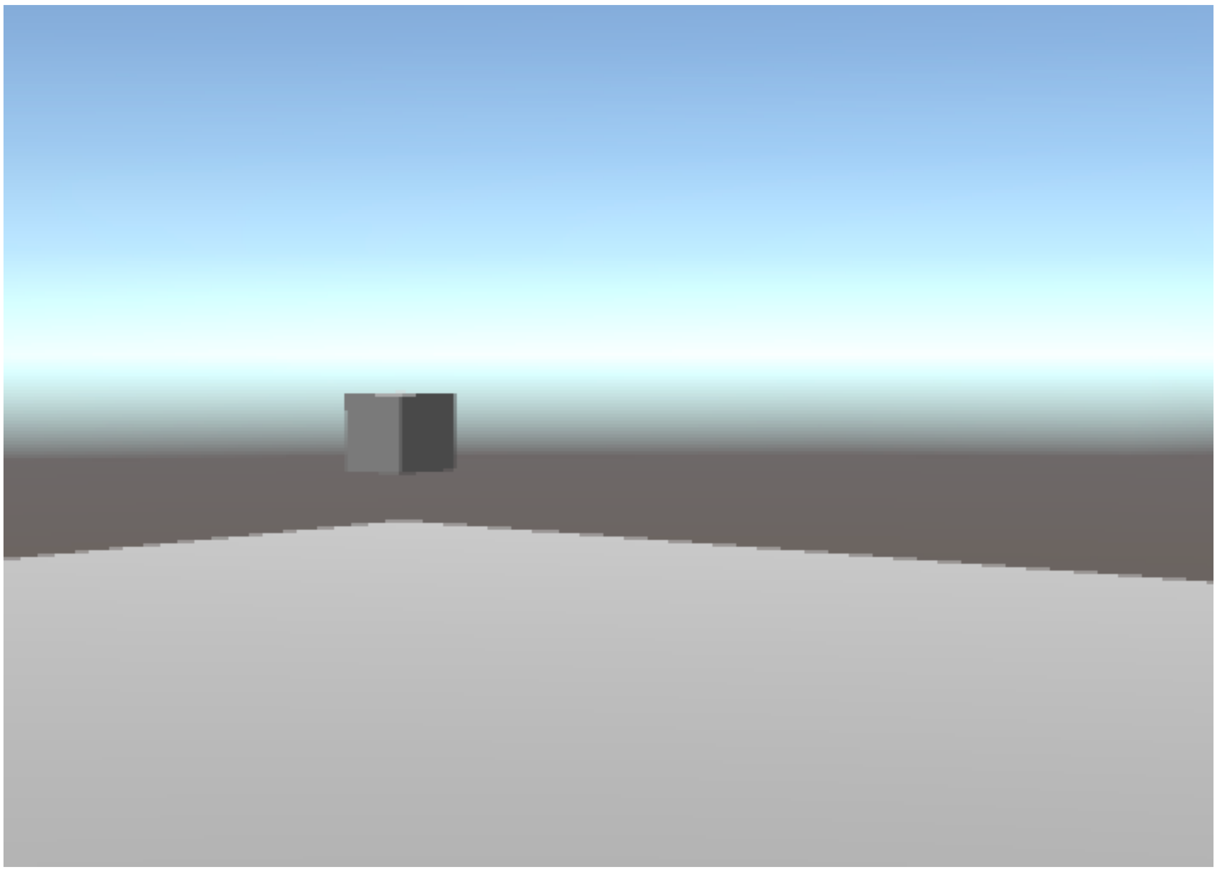
Observations:

- We use a SIP Ping Timer to implement notification sending to start the 'Jump' process
- We duplicated the Robot a couple of times so a whole group is jumping to the jump command. Notice the one robot that does not jump. That is intentional.

Scene 09 – delayed notification

Goal:

Schedule an action to be executed in a few seconds



Challenge:

Low, as without SIP you can do this with Invoke()

Implementation:

We kick a cube upwards a few seconds after 'Jump' is pressed

Scene Controller

```
void Start () {
    subscribeTo ("Jump Pressed");
}

public override void OnNotification (string notificationName) {
    sendNotification ("Kick", 3.0f);
}
```

Start()

Subscribe to the Input event so we know when user pressed Jump

OnNotification()
Schedule a kick notification in three seconds

Cube
```
void Start () {
    subscribeTo ("Kick");
    rb = GetComponent<Rigidbody> ();
}

public override void OnNotification (string notificationName){
    rb.AddForce (Vector3.up * 600f);
}
```

Start()
Subscribe to the kick event

OnNotification()
kick the cube: add a kinetic impulse in the up direction.

Observations

- If you press space multiple times, multiple kick events are scheduled, as expected.
- We use an input Prefab
- Duplicate the cube a couple of times to have a swarm of cubes being kicked.

Scene 10 – Reentrancy

Goal:

Demonstrate that you can (un)subscribe during a notification

Challenge:

Fully reentrant notification managers are a bitch to program correctly. We did it for you.

Implementation

We use a ping repeating timer to regularly send out notifications. At notification time we subscribe or unsubscribe to another notification. If this runs without crashing, the notification manager is fully reentrant (at least for single-threaded scripts)

```
void Start () {
    subscribeTo ("Subscribe");
    subscribeTo ("Level2");
}

public override void OnNotification (string notificationName) {
    if (notificationName == "Subscribe") {
        // we'll re-enter the notifiation manager now
        sendNotification ("Level2");
    }

    if (notificationName == "Level2") {
        // we are a notification inside a notification - the ideal
        // moment to ask for a subscription or to unsubscribe
        // a lesser notification manager will die now. Painfully.
        if (subbedToOther) {
            unsubscribeFrom ("Other");
            subbedToOther = false;
        } else {
            subscribeTo ("Other");
            subbedToOther = true;
        }
    }

    if (notificationName == "Other") {
         Debug.Log ("Other at " + Time.time.ToString ());
    }
}
```

Start()
subscribe to “Subscribe” and “Level2”

OnNotification()
Inside the first level, we alternatively unsubscribe (first challenge to re-entrancy) and notify “Level2” (second challenge to notification). We use a Timing Ping to start the notifications, and another timer for the “Other” notification that gets subscribed to and

unsubscribed. If this script survives successfully more than four cycles, it’s programmed correctly.

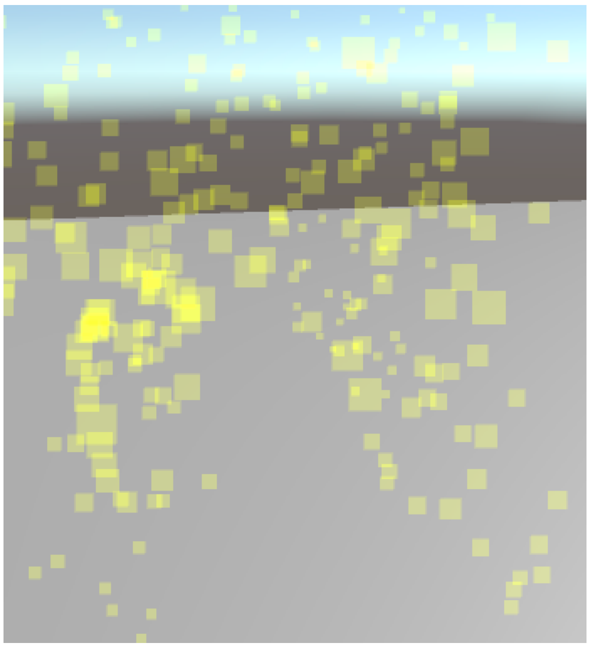
Observations

- SIP's notification manager is also multi-thread safe
- We are using two independent timers for “Other” and “Subscribe” that run on different frequencies.

Scene 11 – particle control prefab

Goal:

Turn particle systems on and off by notification; shows off the Particle Control Prefab



Challenge:

A common asset integration task. We have a Prefab for that.

Implementation

- We use a RemoteParticleControl prefab that turns on/off all particle systems in the Object.

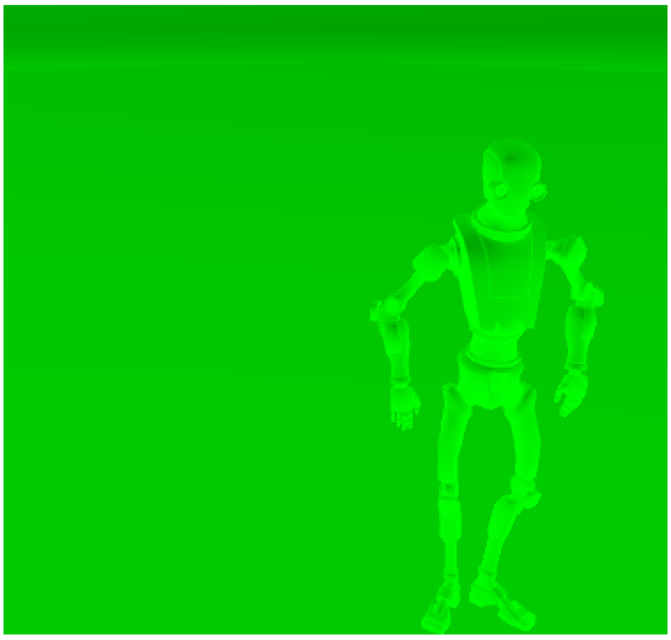
Observation:

- This is just a convenient prefab we produced
- We intentionally use two different notifications for on and off in the prefab.
- Note that SIP Prefabs attach their scripts during runtime only

Scene 12 – multiple simultaneously scheduled notifications

Goal:

Chain sequential actions by scheduling notifications with different information



Challenge:

Scheduling the same action with different data can't be easily done in Unity, as a delayed Invoke() doesn't allow passing data

Implementation

We first fade in a camera effect, and then, a short while later, fade it out again.
We use the Info Dictionary that SIP passes with each notification to pass information that controls fade in and fade out

The script that runs (schedules) the full camera FX (3 steps: fade in, fade out, disable)

```
public void runCameraFX () {
   // 1. Start fx fade in
   Dictionary<string, object> info = new Dictionary<string, object> ();
   info ["FadeIn"] = "fadein";
   info ["Duration"] = "0.25";
   sendNotification ("startFX", info);

   // 2. Start fx fade out, same duration.
   info.Remove ("FadeIn");
   sendNotification ("startFX", info, 1.5f);

   // 3. Stop fx
   sendNotification ("stopFX", 2.0f);
}
```

runCameraFX()

! This demo takes advantage of SIP's capability to pass arbitrary information using the info dictionary. For details, please refer to Geek Sector A section.

To control the fx, we also send additional information in the info dictionary: If the "FadeIn" keyword is present, the camera effect will run a fade-in, else a fade-out. Also, the "Duration" key holds the fade duration. We set up a dictionary accordingly, then we send three notifications:

The first is sent immediately; it turns on the camera fx, and starts the fade (that lasts 0.25 – "Duration")

We also schedule a second f/x in 1.5 seconds from now (1.25 after the fade-in completes). Since we removed the "FadeIn" key from the info dictionary, this is going to be a fade-out. The "Duration" key is still present, so the fade-out will take .25 seconds as well.

The third notification is scheduled in 2.0 seconds, .025 after the fade out completes. The "stopFX" causes the whole camera fx to disable itself, and will no longer burn CPU and GPU cycles.

Observations

- Controlling camera effects with notifications is extremely efficient, and the easiest way to do so from anywhere in your scene – especially when you recall that SIP-controlled script can enable and disable themselves
- We do not discuss the camera script here – please look at it; we guarantee you, it's not that difficult to understand (not counting the shader) and well worth the look, as this is how you can implement fading yourself.
- This script enables and disables the camera effect
- We pass arbitrary information with the notification manager: the keys "Duration" and "FadeIn" are used to pass information that only the methods that subscribe to this notification know about.

Scene 13 – locally delayed notifications

Goal:
Cancel a scheduled notification

Challenge:
prevent a scheduled notification from being sent out

Implementation
There are two ways to schedule a notification: on the notification manager (default), and locally on the script. The latter comes with some preconditions: the object must still exist when the notification is to be sent out, and during scheduling, the script must be enabled. When you can guarantee this, you can tell SIP to schedule the notification locally on the script itself. If you do that, you retain the ability to cancel all currently scheduled notifications on that script (note that SIP does not provide fine-grained notification cancellation: it only allows you to cancel locally scheduled notifications, and then in an all-or-nothing approach)

We use a dissolve shader (provided by cf/x image labs) to first dissolve, and then later resolve (i.e. fade back in) a robot actor. The scripts to dissolve and resolve are directly attached to the models, and are started with a notification provided by the Input prefab.

For this example, when the fade out is complete, the script schedules a fade-in in one second. This is the only scheduled fade, but it can clash with another fade if you start a new fade out after the fade-out has completed. So when the fade-out starts, it cancels all scheduled fades-ins by invoking StopAllCoroutines().

```
    void Start () {
        subscribeTo ("Dissolve");
        subscribeTo ("Resolve");
        mat = GetComponent<Renderer>().material;
        if (fadeTime < 0.1)
            fadeTime = 1f;
    }
```

Start()
Little surprise here as the script subscribes to two notifications: one to dissolve the model, and the other to put it back together.

```
public override void OnNotification (string notificationName) {
    if (fading)
        return;

    if (notificationName == "Dissolve") {
        StopAllCoroutines();
        startFadeOut ();
    }

    if (notificationName == "Resolve") {
        startFadeIn ();
    }
}
```

OnNotification:
In the Dissolve clause, we invoke StopAllCoroutines() to prevent any scheduled fade-ins from starting.

```
void Update () {
    if (!fading)
        return;
    // we are fading. Set shader's dissolve amount
    float amount;
    if (fadeOut) {
        amount = 1 - (currentFadeTime / fadeTime);
    } else {
        amount = currentFadeTime / fadeTime;
    }
    mat.SetFloat ("_DissolveAmount", amount);
    currentFadeTime = currentFadeTime - Time.deltaTime;
    if (currentFadeTime < 0) {
        fading = false; // stop fading altogether
        if (fadeOut)     {
            StopAllCoroutines(); // stop all scheduled notifications
            sendNotification("Resolve", 1.0f, true); // start fade
                        // in in one second, LOCAL post
        }
    }
}
```

Update()

We use Update here for the fade itself. If we are not actively fading, we immediately exit Updte(). Depending on the direction of the fade (controlled with fadeOut), we calculate the amount of fade (a value between 0 and 1, the formula used is essentially a lerp).
We then set the shader's dissolve amount via the material property.
If we have reached the end of the fade and we are fading out, we schedule a fade-in in 1 second) and turn off fading.

Observations

- Better notification cancellation may be part of a future releas
- Use this code to model your own model fades on, including triggering the fade via notification

Scene 14 – sendNotification during Start()

Goal:

Allow your script to send a notification (or run a query for that matter) during Start() (or rather: before the first Update() call). Here we have two scripts that not only subscribe to each other's notifications, but also send each other a notification during start. No matter in which order their Start() methods are invoked, under normal circumstances this will lead to a situation where one notification is lost because the other script hasn't yet subscribed.

Challenge:

During scene start (or whenever you enable/activate multiple objects at the same time), it may become necessary to send a notification or run a query during the script's Start() message. However – especially during the start of a scene – that is the same time when all scripts are signing up for notifications, and it is likely that a notification is lost because not all scripts have subscribed.

Implementation

SIP provides a special DelayedStart() message that is guaranteed to execute before the first Update() is invoked (note: due to limitations of Unity's execution model it will execute after the first FixedUpdate() is invoked).

In order to be able to utilize DelayedStart(), you must observe SIP's own Start() protocol, which you do by prepending 'public override' to your Start() implementation, and also call base.Start() inside your own Start():

```
public override void Start () { // NOTE OVERRIDE!
    base.Start (); // NOTE BASE.START()!
    subscribeTo (subTo);
}

public override void DelayedStart () {
    sendNotification (notifyOf);
}

public override void OnNotification (string notificationName)    {
    Debug.Log ("Received notification: " + notificationName);
}

void Update(){
    if (!updated) {
        Debug.Log ("Update Invoked");
        updated = true;
    }
}
```

If we attach this script to two objects, and cross “A” and “B” for subTo and notifyOf for both objects, the sign up for each others notifications, and at the same time send their own.

Public override void Start()
Note the *public override* and base.Start() changes to Start(). This marks Start as an extension to SIP’s own Start() method (override), and invokes SIP’s mechanism to enable DelayedStart() (done by invoking base.Start()) that will run before the first Update() call. During Start itself we now simply subscribe to the other object’s notification.

DelayedStart()
This method is invoked whenever you enable SIP’s own Start() method as described above. It executes before Update (but after FixedUpdate), and is the perfect time to send notifications or run queries that you would want to run during Start(), but can’t because you may run into race conditions (i.e. must ensure that everyone has subscribed and/or signed up to respond). DelayedStart() is executed once per script, and only if you enable it via override/base.

Update()
We implement this here simply to generate output that shows that the notification was received before Update() was invoked the first time.

Observations

- Using DelayedStart() and splitting subscribeTo from sendNotification into separate Start() and DelayedStart() we can avoid race conditions
- We now can have scripts to subscribe to each other *and* send notifications/queries to each other before the first Update() runs

Scene 15 – networked notifications

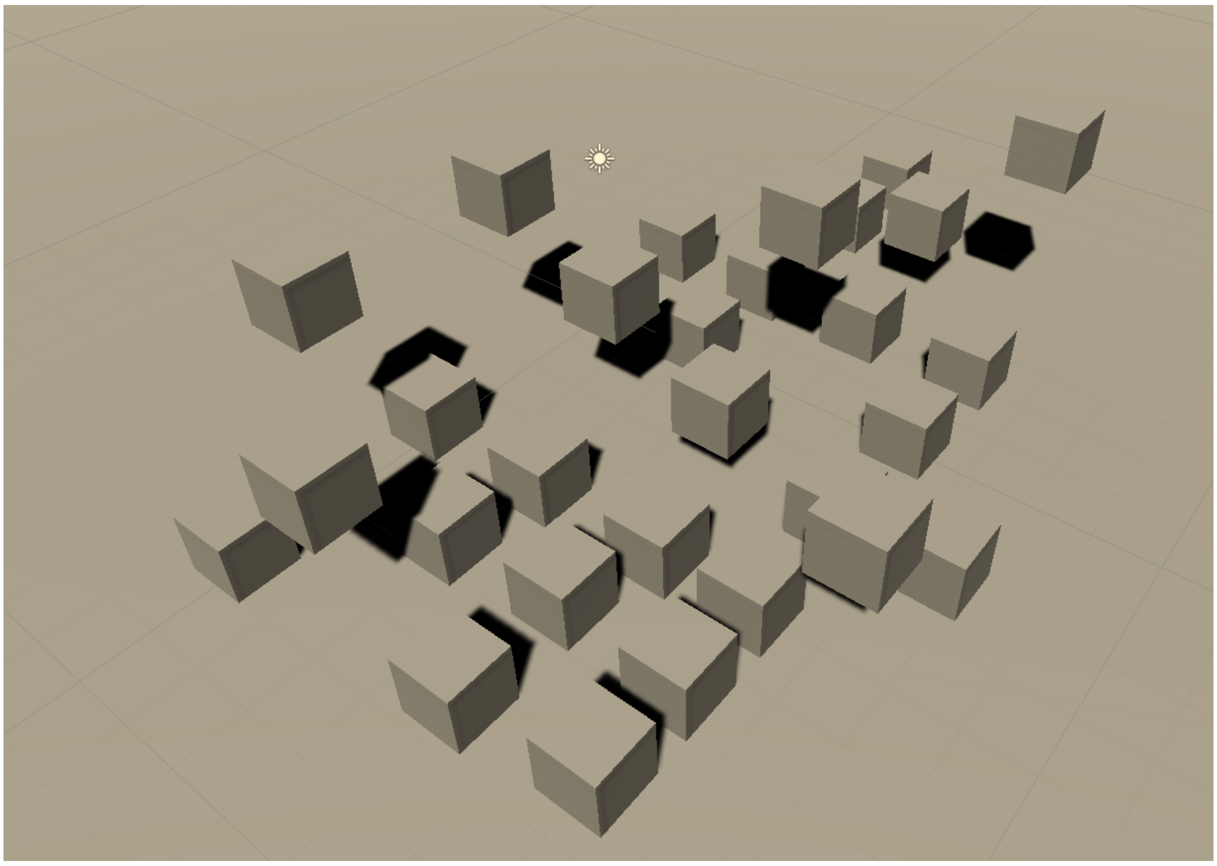
Goal:

Send notifications across the network so every client receives the notification, not just the objects in the local scene.

Scene 16 – sending notifications to myself – in a swarm (Unique Notification Names)

Goal:

Using Notifications, it is very easy to time events, and to use that to control behavior by sending notifications to itself.



Challenge:

Since notifications use names to identify the notification to respond to, we run into a problem when the notification is to be used to control *individual* objects, yet that object has many copies. In a previous example (the 'Robot Jump!' Example), we used a notification to have a lot of objects with identical scripts to respond at the same time. We now want a bunch of objects with identical script to respond individually without needing to change the code or attributes.

Implementation

SIP provides a means to make notification names at the same time script-consistent and scene unique with UniqueNotificationName(). Inside the same script, the result of UniqueNotificationName is always the same during runtime, but across multiple copies of the script in the scene, the result is different, so each copy of the script has their own notification name.

In our example, we simply write a script that after a random interval, we 'kick' the object into the air, and then start a new random interval. We use

UniqueNotificationName() to make the notification name script unique, and can then create multiple copies of the object in the scene without them triggering each other:

```
public override void Start () {
    base.Start ();
    rb = this.gameObject.GetComponent<Rigidbody> ();
    subscribeTo (UniqueNotificationName ("jump"));
    delay = Random.Range (1.5f, 4.0f);
    postNotification (UniqueNotificationName ("jump"), delay);
}

public override void OnNotification (string notificationName){
    if (notificationName == UniqueNotificationName ("jump")) {
        // jump
        rb.AddForce (Vector3.up * 400f);

        // jump again in a random amount of time
        delay = Random.Range (1.5f, 8.0f);
        postNotification (UniqueNotificationName ("jump"), delay);
        return;
    }
}
```

Start()
This method first grabs the object's rigidbody for applying the 'kick' later.
It then uses UniqueNotificationName() to create a script-unique notification name based on 'jump', and subscribes to that unique name
Finally, it posts a delayed notification, again using UniqueNotificationName() to create the same unique name as before based on 'jump'

OnNotification()
We test if the notification is the same as our unique name (note: this is entirely unnecessary, as in this script we already know it to be equal, as otherwise OnNotification would not have been invoked. We wanted to demonstrate that UniqueNotificationName() always returns the same result inside the script), and if so, give our object's Rigidbody a little kick.
We then again post a notification with our unique name after a random delay

After we attach this to an object, we create multiple objects in the scene and let it run.

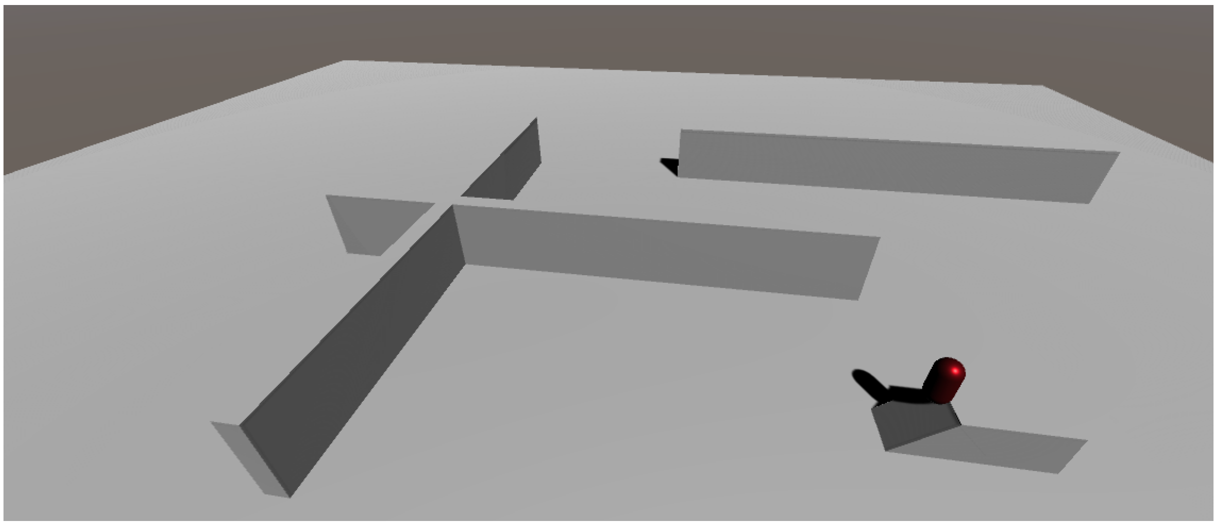
Observations

- UniqueNotificationName() allows you to create individual, script-unique notification names on the fly to enable notification-driven, single-object control inside an object swarm
- There are multiple ways to achieve the same result; using UniqueNotificationName() is merely a convenient one
- You can use the same approach (through UniqueNotificationName()) for event names for some more clarity in your code; performance-wise the code we presented here is slightly faster
- UniqueNotificationName works by simply appending a non-random but script-unique string to whatever you feed into it. You can easily replace that with your own mechanism.

Scene 17 – Navigation (Query & Notifications combined)

Goal:

Build a smart navigation system that allows an actor to collect waypoints from the current scene, and then patrol along those waypoints. The actor should automatically adapt its route to the number of waypoints, meaning that you can change the route layout simply be adding or deleting waypoints to the scene.



Challenge:

Here we have two challenges that are easy to solve with SIP:

- First, we need a way to automatically gather waypoints and sort them
- We also need the actor to detect when it reaches a waypoint, and then proceed to the next

Implementation

We use the Query function for the waypoint objects to identify themselves to the script that controls the actor. The actor collects the waypoints at the start (at delayed start, so all waypoints can sign up before the actor corrals them). And then uses Unity's NavMeshAgent to move the GameObject from one waypoint to the next. Each waypoint uses a trigger (Collider), and when the actor moves into the collider, the waypoint's OnColliderEntry is triggered, causing it to send out a notification. The actor receives the notification, and proceeds to the next waypoint in the list.

Scene Setup

For this tutorial to work we, are using Unity's really nice, and easy to use 'NavMesh', an automatically generated object, that allows Unity's equally nice and easy to use NavMeshAgent to traverse (that's posh for 'move across') the scene. The demo scene consisting of a couple of cube and a plane was set to static, and then we clicked on 'Bake' in the Navigation tab – done. We had more trouble building the ramp, for which we had to go as far as actually turning a cube by 45 degrees!

Waypoint

The waypoint signs up to be queried, and when the query comes, it simply returns its GameObject, so that the actor can build a list of all waypoints in the scene.

```
public override void Start () {
        base.Start ();
        respondToQueryNamed ("waypoint");
}

void OnTriggerEnter(Collider other) {
        // since only one thing moves, we know it's the player
        sendNotification("waypointReached", gameObject);
}


public override object OnQuery (string queryName, string lookFor,
Dictionary<string, object> info) {
        return gameObject;
}
```

Somewhat carelessly, we omit a check for the correct query name in OnQuery, and simply return our gameObject to anyone who queries us. We expect this to be called in the beginning of the scene, when the actor compiles a list of all waypoints.

In OnTriggerEnter, which is called when the Actor moves into our collider, we for simplicity don't check if its really the actor. We just assume it is, and send out a notification "waypointReached", and add our GameObject so, the Actor can identify it, and pick the next one.

Actor

The actor is quite simple: at DelayedStart (because it executes a query and must be sure that all waypoint objects have signed up to be queried this can't be done during Start() itself. Note that you must call base.Start() in order for DelayedStart() to be scheduled), the Actor issues a query to get a list of all waypoints:

```
public override void Start () {
        base.Start ();// if we don't call this, no delayed start!
        agent = GetComponent<NavMeshAgent>();
        subscribeTo ("waypointReached");
}

public override void DelayedStart() {
        // Delayed start so all wp have time to sign up for query
        List<object> queryResults = runQuery ("waypoint");
        waypoints = convertToGameObjects (queryResults, true);

        // access the list of waypoints, and start towards the first
        GameObject theWaypoint = waypoints[0];
        agent.SetDestination (theWaypoint.transform.position);
}
```

One we get a list of all the waypoints, we convert that list of generic objects into a list of GameObjects sorted by their name using one of SIP's exceedingly thoughtful helper functions (no we didn't write those just to look cool in this Tutorial).

For many people not familiar with Unity's built-in AI system, the next part is pure magic, and we wish we could take credit for it: agent.SetDestination() causes Unity to start moving the parent object (via the NavMeshAgent component) towards the desination point – while automatically avoiding obstacles, and correctly moving up ramps. You don't even need an Update() here!

The final part comes during notification. The waypoints are set up so that when the actor moves within the collider sphere, a notification is sent out, to which the actor has subscribed:

```
public override void OnNotification (string notificationName,
GameObject theObject) {
        int newIndex = 0,
            maxNum = waypoints.Count,
            currentIndex = waypoints.IndexOf (theObject);

         newIndex = currentIndex + 1;
         if (newIndex >= waypoints.Count) {
             newIndex = 0;
         }
         GameObject theWaypoint = waypoints [newIndex];
         agent.SetDestination (theWaypoint.transform.position);
}
```

All we do in OnNotification is find out which waypoint we reached, and then pick the next one, wrapping around if we reached the last one. The Demo scene also provides a randomize functionality which we have removed from this discussion.

Observations

- You must call base.Start() in Start() in order to get DelayedStart()
- Here we have a legitimate use of running a query during Start(), which is why we include the DelayedStart()
- You really should look into the NavMeshAgent stuff, you can do really cool stuff with very little effort once you have SIP
- You'll soon find out that Queries and Notifications work great together

Part VII: SIP Integration with Assets

In this section, we demonstrate the various doodads that come with SIP, and how you can use them to quickly enable other assets or old scripts to send notifications that you can use to integrate them.

Utilities – Enhancing existing functionalities

There are several surprising shortcomings in Unity that require a lot of effort from game designers to work around. With some of these, our notification manager provides instant, and surprisingly well rounded, relief.

Then there are third-party assets, that you can't (and shouldn't, really) easily modify yourself. Getting these to work with the notification manager is usually quite simple (provided the asset is well-designed, and your needs reasonable). We have provided some integration prefabs or scripts that retro-fits these objects with notification capability literally at the drop on a prefab.

!

FEATURE ALERT – DO NOT IGNORE
The integration prefabs have been engineered to **attach** themselves **to** the **parent** object **during runtime only**. This is to ensure that you can quickly remove the whole prefab to restore original functionality, keep the original asset's integrity, and to be able to quickly find any notification prefab in your scene.

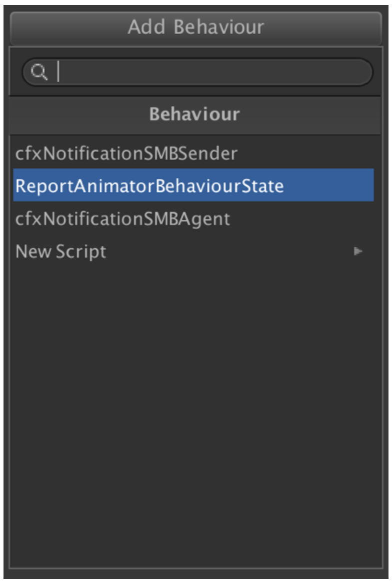
For integration, simply drop the prefab on the object you want to integrate.

The Animator State Conundrum

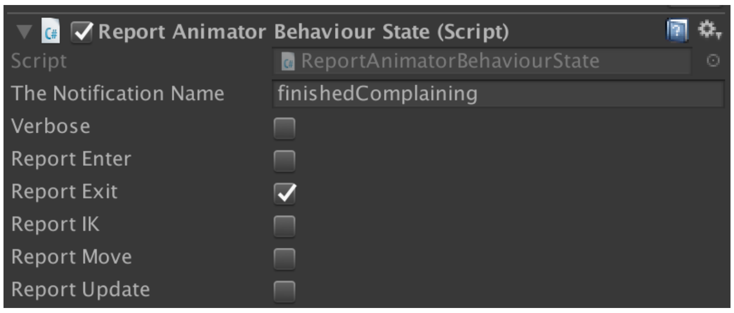
One oversight in unity (at least up to and including 5.6) is that main scripts don't know, nor are notified, when your Animator's FSM changes state. This isn't a problem for 90% of the time, but unfortunately, almost every game (at least those that use an AnimationController) need to know when a key animation has begun or finished. There are multiple ways to tackle this, and the best would be if Unity provided an OnAnimatorStateChange method, which they don't. Well, if you have the notification manager, we provide you with a nifty little script that integrates this into your code.

If there is a state in your FSM that you need to know about (be it entered, exited, updated, IK or moved), simply attach a ReportAnimatorBehaviourState script from the menu, enter a name for the notification, and you are good to go.

One additional boon of this notification script is that it gives you free access to the active Animator, a component that is notoriously difficult to access from the outside.



You then can configure the properties from the editor, or extend the script yourself.



Report Animator Behaviour State (Script)

Script	ReportAnimatorBehaviourState
The Notification Name	finishedComplaining
Verbose	☐
Report Enter	☐
Report Exit	✓
Report IK	☐
Report Move	☐
Report Update	☐

where

- Report Enter
  Send a notification with <The Notification Name> (in this example "finishedComplaining") when the behavior state is entered
- Report Exit
  Send a notification with <The Notification Name> when the behavior state is exited
- Report IK
  Send a notification with <The Notification Name> when the behavior state has an IK (Inverse Kinetic) event
- Report Enter
  Send a notification with <The Notification Name> when the behavior state reports a move
- Report Updat
  Send a notification with <The Notification Name> when the behavior state reports an update

On notification, the prefab also reports the following context in the info dictionary:

"Event": "EnterState" / "ExitState" / "UpdateState" / "MoveState" / "IKState"
"Animator": the Animator that is currently animating
"StateInfo": the Stateinfo for this state

“LayerIndex”: the index of the Layer this state belongs to
“GameObject”: the object that the audio source belongs to
“Time”: the Unity time (as string) when the notification occured

In our example, we needed to know when our actor finished his complain animation to trigger a response from the other actor’s AI. Once it’s finished, it sends out a “finishedComplaining” event.

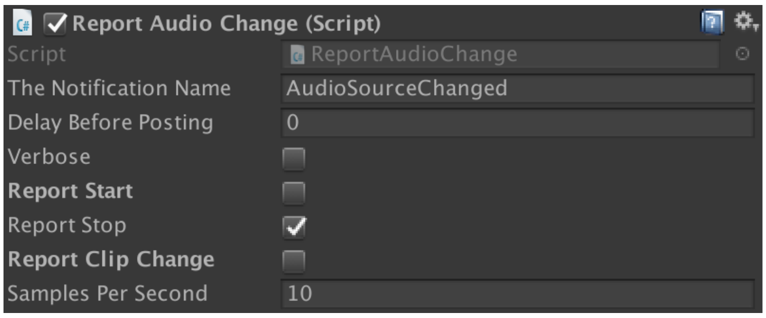
The AI for the *other* actor (that has no link to the complaining actor’s object) picks up the notification, and responds by initiating an animation, and then posting it’s own event to tell the scene that it has responded.

```
public class punchAI : cfxNotificationIntegratedReceiver {

    // Use this for initialization
    void Start () {
        // the notifier script in the animator will send this event
        // when the model completes the 'Angry' animation
        subscribeTo ("finishedComplaining");
    }

    public override void OnNotification (string notificationName)
    {
        // tell my animator to start the 'punch' animation
        Animator anim = gameObject.GetComponent<Animator> ();
        anim.SetTrigger ("punch");

        // send out notification that I just have thrown a punch
        sendNotification ("thrownPunch");

    }

}
```

You see that notifications are perfect for synchronizing animations, and this alone justifies using a notification manager.

The Audio Source Problem

Like with the animator, Unity doesn't provide robust notification when an Audio Source has finished playing a clip. Sure, you can poll (which is the accepted solution) until you observe a change in status. It's astonishing that something as basic as this doesn't have better support.
Our notification manger can only provide a somewhat better solution here, but it at least harmonizes the flow of control. Add the ReportAudioSourceChange prefab to the object that contains your audio source (see the feature alert above!), and tell it what event to look for. As soon as the state of the audio source changes, the notification is sent, and your script can react.


Report Audio Change (Script)

Script	ReportAudioChange
The Notification Name	AudioSourceChanged
Delay Before Posting	0
Verbose	☐
Report Start	☐
Report Stop	✓
Report Clip Change	☐
Samples Per Second	10

The important properties here are

- Report Start
  Send a notification when a clip starts

- Report End
  Send a notification when the clip ends

- Report Clip Change
  Send a notification when the clip changes. For reasons known only to Unity, this does not apply to clips played with AudioSource.playOneShot().

- Samples Per Second
  We must use a form of busy wait to determine all changes. We implemented this via a fast co-routine that runs n times per second. 10 is usually enough for all games.

On notification, the prefab also reports the following context availanle in the info dictionary under the following keys:

"Event": "Start" / "Stop" / "ClipChange"
"Audio": the Audio clip's name. Does not apply to clips played with playOneShot
"GameObject": the object that the audio source belongs to
"Time": the Unity time (as string) when the notification occured

So, let's whip up some code to use this:

```
public class speakerofthehouse : cfxNotificationIntegratedReceiver {

    public AudioClip monkey1;
    public AudioClip monkey2;

    public AudioSource theSource;

    private AudioClip playing;

    void Start () {
        subscribeTo ("AudioSourceChanged");
        // now start the first clip
        theSource.PlayOneShot (monkey1);
        playing = monkey1;
    }
    public override void OnNotification (string notificationName) {
        // Ping-Pong between clips
        AudioClip nextClip;
        if (playing == monkey1) {
            nextClip = monkey2;
        } else {
            nextClip = monkey1;
        }
        theSource.PlayOneShot (nextClip);
        playing = nextClip;
    }
}
```

In our own example above, we simply want to play through a list of audio tracks – as you usually do in a tutorial, where you have to ensure the player has progressed enough in order to play the next audio clip. Here we merely alternate between two audio clips, but you can substitute your own code that decides what clip to play when.

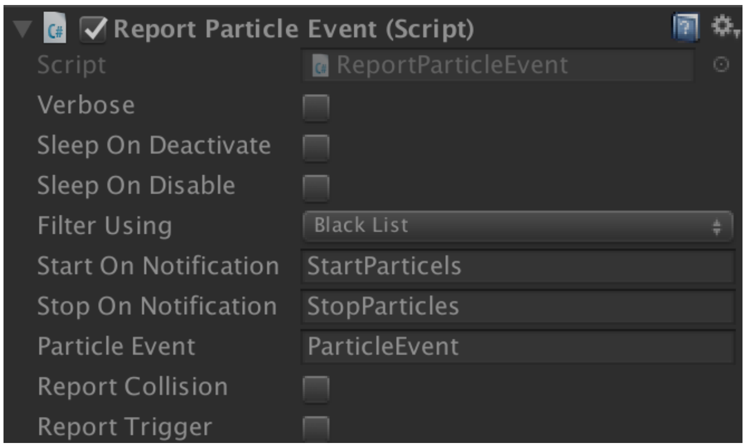
!	Due to some design ~~flaws~~ choices in Unity's audio component, even our monitoring script employs polling, wasting some CPU resources. However, we employ a co-routine that uses much less CPU than an Update(), and you can set the number of polls taken – usually 20 polls per second is more than enough.

Particle Systems Prefab

This little prefab is a near-perfect case study of what a notification manager can do. The prefab subscribes to two notifications (named by you), and upon receiving the notification turns the particle emitters of all particle systems inside the object (i.e. all children) on or off (i.e. starts or stops them).

Furthermore, if a particle of the main particle systems collides or triggers, the prefab sends out an event.

Like all our prefabs, the script is attached during run-time only, so you can easily add and remove it to third-party assets.



Report Particle Event (Script)	
Script	ReportParticleEvent
Verbose	☐
Sleep On Deactivate	☐
Sleep On Disable	☐
Filter Using	Black List
Start On Notification	StartParticels
Stop On Notification	StopParticles
Particle Event	ParticleEvent
Report Collision	☐
Report Trigger	☐

- Start On Notification:
  The notification name for a notification another script sends out to activate the particles. If no notification name is given, the prefab will not listen for start notifications. This name can't be changed at runtime.

- Stop On Notification:
  The name for a notification that another script sends out to deactivate the particle system. If no notification name is given, the prefab will not listen for stop notifications. This name can't be changed at runtime.

- Particle Event:
  The notification name that the prefab sends out when a Collision or Trigger particle event happens. You subscribe to this name. You can change this at runtime.

- Report Collision
  Controls if a notification is sent out after a particle collision

- Report Trigger
  Controls if a notification is sent out after a particle trigger event

The following entries are defined in the Info dictionary that comes with the notification:

"Event": "Collision" / "Trigger"
"Other": (Only collision) the object that the particle collided with
"GameObject": the object that the particle emitter belongs to
"Time": the Unity time as string when the notification occurred

A short discussion on design of this Prefab

This prefab uses two different notifications to turn particles on and off. Wouldn't an implementation with a single notification (e.g. "ParticleState") and two different events ("On" and "Off") be more efficient and elegant?

Yes.

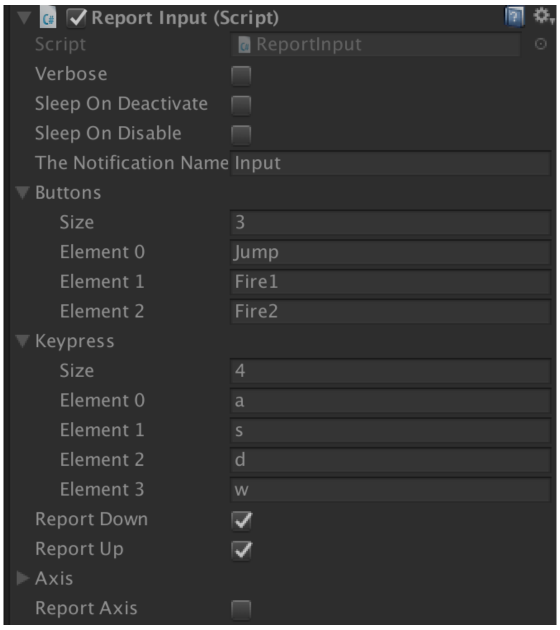
That being said, though, we designed a *Prefab* here. Prefabs are handled in Editor by the user and if we used a single notification with two states, this would have required the user to change three public variables instead of two.
This makes the integration process more cumbersome, and hence we decided to implement the Prefab with two different notifications.

Input Notification (standard and advanced)

Initially, you won't find this utility very helpful, but once you start realizing just how much more efficient you and your scripts become once you switch to a notification event-based model, you'll appreciate just how convenient this script is. This is one of the few notification utilities that latch into Update(), to look at the Input Manager. Placing this little object anywhere in your scene, gives all your scripts the ability to be notified whenever a certain key or button is pressed – without needing to implement Update() themselves

There are multiple versions of this prefab, with varying features – some of which require editor enhancements like "Odin" (no affiliation with cf/x) to use efficiently.

Be advised that these prefabs must be configured to match your Input Manager configuration; this is not an automated process.



Report Input (Script)	✓
Script	ReportInput
Verbose	☐
Sleep On Deactivate	☐
Sleep On Disable	☐
The Notification Name	Input
Buttons	
Size	3
Element 0	Jump
Element 1	Fire1
Element 2	Fire2
Keypress	
Size	4
Element 0	a
Element 1	s
Element 2	d
Element 3	w
Report Down	✓
Report Up	✓
Axis	
Report Axis	☐

- The Notification Name
  Name of the notification the Prefab send out when it detects input. You must subscribe to this name to receive a notification.

- Buttons
  A list of button names (as defined in your Input Manager) that the Prefab should monitor.

- Keypress
  A list of keys (on your keyboard) that the Prefab should monitor.

- Report Down:
  If set, the prefab sends out a notification when a button or key in the 'Buttons' or 'Keypress' lists is pressed. You can access which button/key is down from the Info dictionary (see below).

- Report Up:
  As above, except for the 'Up' event.

- Axis:
  A list of axes (e.g. Horizontal, Vertical, Thrust, whatnot) that – again according to your Input Manager – are available to your game and should be monitored for change.

- Report Axis
  If set, the Prefab sends out a stream of notifications for any axis defined in the list above. Please note that this is terribly inefficient, and should only be used for prototyping or testing purposes.

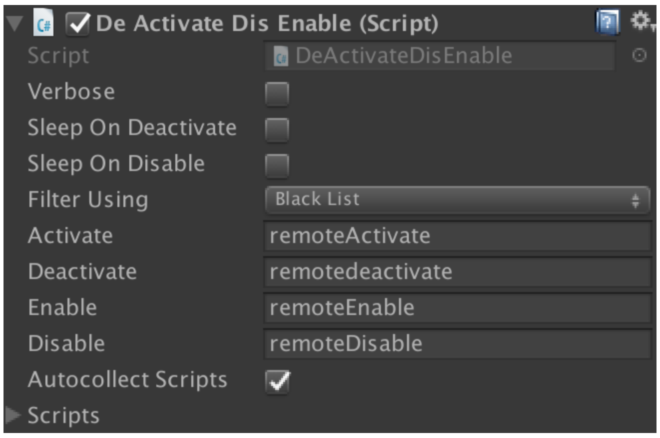
The following information is available from the Info dictionary when your script receives a notification:

"Event": "ButtonDown" / "ButtonUp" / "KeyDown" / "KeyUp" / "Axis"
"Button": Only on ButtonDown or ButtonUp: the name of the button
"Key": Only on KeyDown or KeyUp: the name of the Key
"Axis": Only on Axis: the name of the Axis
"GameObject": the object that this prefab is attached to
"Time": the Unity time (as string) when the notification occurred

We have provided a more advanced version of the prefab that provides separate notification names for separate key or button press/release actions. Since Unity can't display Dictionaries in it's default editor (unless you resort to editor enhancements), we leave it to you to explore this Prefab the hard way. Just rest assured that it's a really cool Prefab.

Remote Enable / Disable / Activate / Deactivate

This is a simple prefab to allow you to remotely enable/disable scripts inside assets, or activate/deactivate the object. Its main use is for easy integration of third-party assets where you can enable or disable whole groups of objects with a single notification.


De Activate Dis Enable (Script)

Script	DeActivateDisEnable
Verbose	☐
Sleep On Deactivate	☐
Sleep On Disable	☐
Filter Using	Black List
Activate	remoteActivate
Deactivate	remotedeactivate
Enable	remoteEnable
Disable	remoteDisable
Autocollect Scripts	✓
Scripts	

It provides the following controls:

- Activate
  The notification name to send if you want to activate the GameObject

- Deactivate
  The notification name to send if you want to deactivate the GameObject

- Enable
  The notification name to send to enable all the scripts listed in the 'Scripts' property

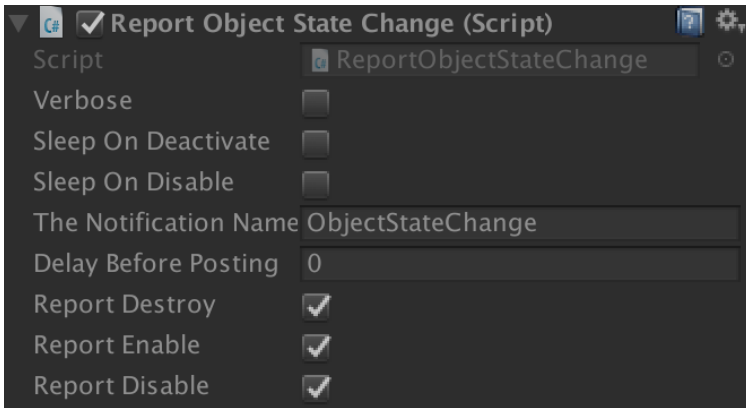
- Disable
  The notification name to send to disable all the scripts listed in the 'Scripts' property.

- Autocollect Scripts
  If checked, Enable and Disable will automatically gather all scripts attached to this GameObject and enable or disable them. Note that this usually includes this prefab's script, but since SIP scripts execute even if disabled, it makes no difference.

- Scripts
  An array that contains the links to the scripts that should be enabled/disabled.

Object State

This is a general purpose prefab that you'll probably base most of your own developments on (should you use SIP-derived Prefabs).



Report Object State Change (Script)

Script	ReportObjectStateChange
Verbose	☐
Sleep On Deactivate	☐
Sleep On Disable	☐
The Notification Name	ObjectStateChange
Delay Before Posting	0
Report Destroy	✓
Report Enable	✓
Report Disable	✓

It provides the following controls:

- The Notification Name
  Name of the notification the Prefab send out when it detects an object state change. You must subscribe to this name to receive a notification.

- Delay Before Posting
  Number of seconds between the moment the Prefab detects the change and sends the notification. The Prefab can detect and delay multiple changes at the same time.

- Report Destroy
  Send a notification when the object is destroyed. This will always be sent out immediately, ignoring the Delay Before Posting parameter. All pending notifications are discarded.

- Report Enable
  Sends a notification when the object is enabled

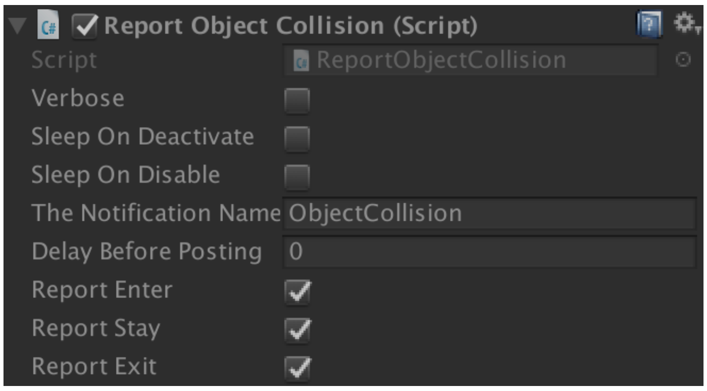
- Report Disable
  Sends a notification when the object is disabled

The following entries are defined with the notification:

"Event": "Destroy" / "Enable" / "Disable"
"GameObject": the object that the particle emitter belongs to
"Time": the Unity time (as a string) when the notification occurred

Collider

One of Unity's best and most critical features are their colliders. Knowing if, when and where two objects touch is a central task for most games. Also, most game designs rely on third-party assets, as they are an economic way to quickly build a new game out of proven blocks. A classic task, therefore, is integrating hit detection into third party assets (some designers may say it's almost 50% of their time). The notification manager provides a prefab that you simply can drop into a third-party asset, and it will monitor that object's collider and send a notification to all interested scripts when a collision occurs (with all relevant collision information included). This allows you to centralize the control of all third-party assets within a single script, with little to no overhead.


Report Object Collision (Script)

Script	ReportObjectCollision
Verbose	☐
Sleep On Deactivate	☐
Sleep On Disable	☐
The Notification Name	ObjectCollision
Delay Before Posting	0
Report Enter	✓
Report Stay	✓
Report Exit	✓

- The Notification Name
  Name of the notification the Prefab send out when it detects a collision. You

must subscribe to this name to receive a notification.

- Delay Before Posting
  Number of seconds between the moment the Prefab detects the change and sends the notification. The Prefab can detect and delay multiple collisions at the same time.

- Report Enter
  Send a notification when an object enters the collider

- Report Stay
  Send notifications as long as an object remains inside the collider

- Report Exit
  Send a notification when an object vacates the collider

The following entries are defined in the Info dictionary that comes with the notification:

“Event”: “Enter” / “Stay” / “Exit”
“Collision”: the Unity.Collision class object that describes all the collision details
“GameObject”: the object that the particle emitter belongs to
“Time”: the Unity time (as string) when the notification occurred

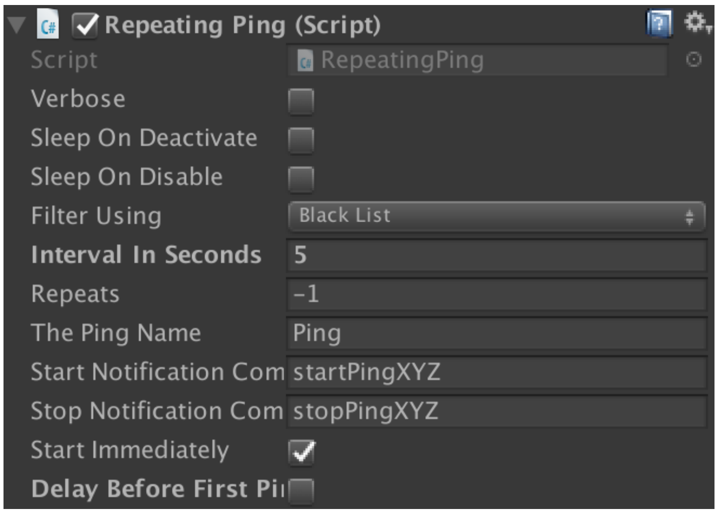
Trigger

[similar to colliders, still in development]

Xxxx show example
Xxxx explain properties

Repeating Ping / Timer

Repeating Ping is a simple timer that you can start and stop with notifications and that can repeat, providing you with a simple, convenient clock, or beacon, that you can use to synchronize actions in your scene.


Repeating Ping (Script)

Script	RepeatingPing
Verbose	☐
Sleep On Deactivate	☐
Sleep On Disable	☐
Filter Using	Black List
Interval In Seconds	5
Repeats	-1
The Ping Name	Ping
Start Notification Com	startPingXYZ
Stop Notification Com	stopPingXYZ
Start Immediately	✓
Delay Before First Pi	☐

- The Ping Name
  Name of the notification the Prefab send out when the timer reaches zero. You must subscribe to this name to receive a notification.

- Interval In Seconds
  The time in seconds to count down from

- Repeats
  The number of times the timer repeat. If you set this to a negative value, it repeats endless. If you set it to zero, the timer will count down exactly once, and will not repeat.
  This means that the amount of Ping notifications emitted from this prefab is always 1 + (Repeats) – unless Repeats is negative, in which case it is until you end the scene.

- Start Notification Command
  The prefab subscribes to this notification. When received, it starts the timer.

- Stop Notification Command
  The prefab subscribes to this notification. When received, it will no longer re-start the timer (this means that the currently running timer is not interrupted and will fire when done. Use the “FullStor” event for that (see below)).

! If you also supply the “Event” entry with a value of “FullStop” in the info dictionary that accompanies the Stop Notification Command, the currently running timer is also stopped.

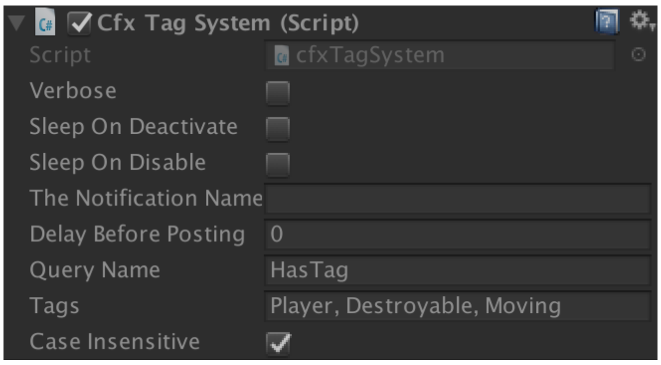
- Start Immediately
  If checked, the Prefab starts the timer as soon as it wakes up.

- Delay before first Ping
  If checked, the Interval In Seconds is observed before the very first ping. If you uncheck this, the Prefab produces the first ping immediately upon staring the timer.

Multi-Tag

The multi-tag Prefab is another showcase for the versatility of a notification and query system. This little gem allows you to give multiple tags to any object, and then use the query system during run-time to corral objects depending on the tag you are looking for.

This prefab is intended as a case study and stepping-off stone for more elaborate prefabs, curtailed to your requirements.



Cfx Tag System (Script)

Script	cfxTagSystem
Verbose	☐
Sleep On Deactivate	☐
Sleep On Disable	☐
The Notification Name	
Delay Before Posting	0
Query Name	HasTag
Tags	Player, Destroyable, Moving
Case Insensitive	✓

- The Notification Name
  Not used.

- Delay Before Posting
  Not used.

- Query Name
  The prefab signs up to answer this query

- Tags
  A simple string where you can enter your Tags. The current implementation simply looks for a substring in the tags, so make sure the Tags are unique and do not contain substrings of other Tags. If, for example, a Prefab contains the tag “Destroyable”, and a query is looking for the Tag “Able”, the Prefab will respond (if case sensitivity is off).

- Case Insensitive
  Ignore case when looking for a tag

A more elaborate version of this prefab that uses a convenient List<string> as tags instead of a string is also included. It can be freely mixed with this prefab for queries in the same scene.

Other Helpful Utilities included with SIP

There are several additional helpful items included with SIP that you might find interesting for your own projects:

Two Color Camera Effect

This camera effect will map every color on your screen to a dissolve between two colors of your choice, and fade the result back with the original image. We use it in one of our demo scenes to demonstrate how to time two successive events to turn the effect on, and then off again.

You can use this camera effect for

- Fading the screen to any color (including the all-time favorites white and black) by setting Left and Right to the same color. We use this effect with a quick cross-fade to red to make the whole screen red when the player is injured.
- Simulate “Night-Vision” (use black as ‘Left’ color, and bright green as ‘Right’

Notification-enabled Camera Script

This script is the counter-part for above camera effect and demonstrates how you can perform fades to any color (and back), script activation and deactivation and control of effect duration via notification

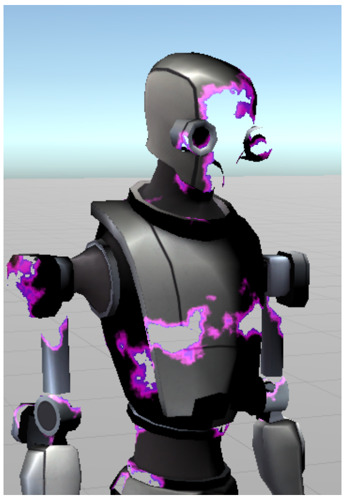
Simple Dissolve

A high-quality dissolve shader that you can use to dissolve objects with. We originally included this shader for the Local Scheduling demo, but subsequently came up with a better looking dissolve shader (also included), that adds a bit more pizzazz to the scene.

Use at your own discretion.

Burn-through Dissolve

This very popular effect dissolves an object, with colored edges that mimic a ‘transporter’ or ‘magic burn’ effect before the parts dissolve. We use it in one of our demo scenes to demonstrate how to schedule and cancel local notifications.



You can use this effect any time you need a good-looking dissolve of an object that was destroyed. To make it look like the object is burning and then dissolving, exchange the black-purple-blue ramp with a black-orange-yellow ramp of your own creation.

Camera Control

This is a small script that you can attach to your camera during early development. The script allows you to move in your Game View the same way that you move in your scene editor (A, S, W, D, E, Q and mouse look when depressing mouse 2). Simply add it to your camera

Caption Prefab

Quickly add a short permanent caption to your scene. Using notifications, you can change this prefab within seconds to display dynamic text. If you read the documentation until this point, you already know how.

Integration Case Study

We have found that since Unity has no built-in notification manager, many first-rate assets must roll their own mechanism. Using the provided cfxNotificationIntegratedReceiver class, you can easily adapt third-party assets without, or with very little effort, and often without code changes inside the assets (this is important when the asset providers update their asset, possibly overwriting any changes you have made).

In the case study below, we adapt one of the finer third party assets in the store, "Kripto's Realistic Effects 4", to work with notifications [please note that we have no affiliation with Kripto, except that we are using their assets and like them very much].

This effects pack contains a number of great-looking "magical" effects, among them a magic missile spell. After you start the effect, it propagates through your scene until it hits something. To be notified of a hit, the documentation recommends that any script that wants to know about the hit, install a callback.

Although this is a workable solution, it is difficult to implement, and lends itself to errors when you move the code around, or when more than one scripts need to know about a hit. Finally, all scripts have to install callbacks to all missiles that are fired, leading to an almost guaranteed memory leak.

A simpler approach is to attach a small notification script to the missile itself that latches onto the same object's hit callback. When the callback is executed, the script connects to the notification manager, and posts the hit information. The notification manager then distributes this notification to all interested parties. The notification script dies with the missile object, and no memory is leaked.

```
public class HitNotifier : cfxNotificationSender {

    public string theNotificationName = "SpellHit";
    public string theSpell; // put name of spell here

    // Use this for initialization
    void Start () {
        var tm = theFX.GetComponent<RFX4_TransformMotion>();
        if (tm != null) {
            tm.CollisionEnter += Tm_CollisionEnter;

        } else {
            Debug.Log("NO COLLISION CALLBACK IN " + theFX.name);
        }
    }


private void Tm_CollisionEnter(object sender,
RFX4_TransformMotion.RFX4_CollisionInfo e) {
        Dictionary<string, object> theDict = getBasicInfo(theSpell);
        theDict.Add("Source", gameObject);
        theDict.Add("Target", e.Hit.transform.gameObject);
        postNotification(theNotificationName, theDict);    }

}
```

Discussion

Most of the magic (pun intended) happens in the first line: the script inherits from cfxNotificationSender (which is a slimmed-down version of the cfxNotificationIntegratedResponder. It has fewer code amenities, and you could use either. We chose it because

a) to show that we can, and
b) it is more efficient. Invisible to this script, by inheriting from a SIP class, we automatically also get the scripts ability to connect to the notification manager as soon as the script activates. All we need this script to do is send an event when the effect calls the callback.

```
void Start ()
```

Here comes the ugly stuff – this is how C# implements what it calls delegation (technically it's call-forwarding, not delegation at all – but we digress). It's anything but intuitive to use, and well worth shielding normal users from. Kripto use this approach not because Andrey is a bad coder, but, short of a notification manager, he doesn't have a better choice. We do now.

```
      tm.CollisionEnter += Tm_CollisionEnter;
```

What this line does is to pass a method to the effect that the effect should call when it determines a hit, using a C# mechanism called 'delegation'. It tells the delegation object to also invoke "Tm_CollisionEnter" when it invokes all the other delegates. Fugly stuff.

```
private void Tm_CollisionEnter(object sender,
RFX4_TransformMotion.RFX4_CollisionInfo e)
```

This is the counterpart to the other ugly thing above. This method is called from the outside when the effect determines it has hit something. We use this to hook in the notification manager:

```
      Dictionary<string, object> theDict = getBasicInfo(theSpell);
      theDict.Add("Source", gameObject);
      theDict.Add("Target", e.Hit.transform.gameObject);
```

we use the notification manager's ability to pass structured data with the event to add information about what was hit to the notification.

```
      postNotification(theNotificationName, theDict);
```

That's all there is. From now on, ANY script in the scene knows when this effect hits something, and can initiate their own activities. Attaching this scrip to the effect's prototype enables you to detect a spell hit in all your scripts, without ever installing a callback, or even knowing about the effect and its method of communicating a hit. Which, incidentally, enables us to unify hit handling – something that makes integrating another great third-party effects package (that uses its own hit-callback mechanism) so much easier

Part VIII: Reference

This section lists the various convenience functions and documents the ‘raw’ interface provided by SIP. This reference section assumes that your script is a descendant from the `cfxNotificationIntegratedReceiver`

Notifications

Notifications are signals that your scripts can broadcast, and other scripts (or even the same script) can receive. Notifications are a voluntary co-op system, so objects that want to receive a notification must sign up and can resign any time. If you sign up to be notified via the provided functionality, you should implement an OnNotification method in your script.

! **Notifications only work if there is a cfxNotificationManager prefab in your scene.**

Information passed with Notifications

All notifications, when sent to scripts are always accompanied by a Dictionary that can contain any additional data you may want to pass to the receiving scripts. It is up to you to define, include and interpret that information.
If you are using SIP’s convenience functions, a number of keys are always present in this dictionary that can be used to retrieve additional data:

- “Event”
  A string you can either set yourself or this is set automatically by SIP. It is used for filtering, and to easily sub-class a notification
- “Time”
  Unity Time as a string when the event was posted. For scheduled events, this is the time the event was originally scheduled.
- “GameObject”
  The game object that the script belonged to when the event was posted. The object may no longer exist if the notification was scheduled. To see if this notification was scheduled, check for the presence of the “Scheduled” key.
- “Scheduled” (only present in scheduled notifications)
  If this key is present, the notification was scheduled. The value is either “Central” if scheduled via notification manager, or “Local” when scheduled locally. If this key is present you should not assume that the scheduling GameObject still exists.

Subscribing to Notifications

You can subscribe to multiple notifications at the same time.

To subscribe to a notification use:

```
void subscribeTo(string notificationName)
```

Subscribes your script to a notification with the name <notificationName>. The variant of OnNotification you implemented in this script will be invoked whenever someone sends a notification that matches <notificationName> exactly.

```
void subscribeTo(string notificationName,
cfxNotificationManager.notificationHandler theHandler)
```

Subscribes your script to a notification with the name <notificationName>. The handler you specified will be invoked whenever someone sends a notification that matches <notificationName> exactly. You must ensure that the handler matches the notificationHandler pattern.

Unsubscribing from Notifications

```
void unsubscribeFrom (string notificationName)
```

Removes <notificationName> from the list of notifications that the NotificationManger invokes your script for. Note that this will only affect the script that is calling the unsubscribeFrom methods. All other scripts that have subscribed to the same notification will still be invoked and have to cancel their subscriptions themselves. Furthermore, any other notification this script is subscribed to remains active.

```
void unsubscribeAll()
```

Removes all notifications that this scrip has subscribed to. Limited only to the invoking script.

Custom Information that you can pass with Notifications

No matter what method you use to send or receive notification, under the hood SIP always passes a lot of information along with your notification. The convenience functions we provide usually mask this information, as it is not required for the majority of tasks. That being said, you can tremendously improve the usefulness of your notifications by passing your custom information with notifications.

SIP provides strong support for this via

- OnNotification() variants with direct access to info
- sendNotification() variants with direct access
- Helper methods to add and retrieve data from info

Please see Geek Sector A for detailed information.

Filtering Notifications

Filtering processes specific data present in the "Event" entry in the info dictionary. If you send notifications that do not fill the "Event" entry automatically, you must ensure yourself that this data is present by setting it with some code similar to

```
info["Event"] = "Something";
```

Otherwise, filtering might behave erratically.

Filtering will be according to the setting of the filterUsing public variable that you can set with script and in the Unity editor. The possible values are WhiteList and BlackList.

```
void filterEventForNotification(string theEvent,
string theNotificationName)
```
Adds theEvent to the list of Events that are to be filtered (white/black) for the notification with name notificationName.

```
void filterEventForNotification(List<string> theEvents,
string theNotificationName)
```
Adds a list of event names to be filterd to theNotificationName for this script.

```
void removeEventFilteringForNotification(
string theEvent, string theNotificationName)
```
Removes theEvent from the filter list for theNotificationName for this script.

```
void removeEventFilteringForNotification(
List<string> theEvents, string theNotificationName)
```
Removes a list of event names from the notification names theNotificationName for this script.

OnNotification Variants

If you sign up to receive notifications using subscribeTo without providing your own method, SIP routes all notifications to a chain of OnNotification methods, of which must override exactly one (not zero, not two; five is right out) in your script.

! YOU MUST NOT CALL THE BASE METHOD IN YOUR OVERRIDE.
When you override one of the OnNotification() variants, make sure not to call the base method you are overriding, as this will – at the very least – cause SIP to log an incorrect warning; in many cases it produces unpredictable results.

Choose the variant that provides the least amount of data you need, as it greatly simplifies your code.

Note that filtering is always applied to all OnNotification methods, regardless if your implemented invocation uses "theEvent" or not.

```
override void OnNotification(string notificationName)
```
Provides simply the notification name. Enough for many cases

```
override void OnNotification(string notificationName,
string theEvent)
```
Provides the notification name and the value for the “Event” key in the info dictionary.

```
public virtual void OnNotification(string notificationName,
GameObject theObject)
```
Provides the notification name and the value for “TheObject” key in the info dictionary.

```
override void OnNotification(string notificationName,
Dictionary<string, object> info)
```
Provides the notification name and the Info dictionary that contains all supplementary information that accompanies the notification.

```
override void OnNotification(string notificationName,
string theEvent, Dictionary<string, object> info)
```
As above, but with the contents of the “Event” entry copied to theEvent, saving you a full line of code.

Implementing your own OnNotification

If you want to have your own method invoked for a notification rather than using OnNotification, you need to subscribe to the notification manager by using the variant that also passes the method. To be able to pass a method, it must match the following pattern:

```
void myMethodName(Dictionary<string, object> info)
```

When invoked, the info dictionary contains all relevant information, and is guaranteed to contain at least entries for the following keys:

- “NotificationName”: notification name
- “Event”: the event or “cfxGenericNotification”
- “Time”: Unity time (seconds since starting) as string
- “GameObject”: the GameObject the notifying script belongs to. Note that this is not necessarily the root object.
- “Scheduled”: only present if the notification was scheduled. In that case it can contain two values: “Central” or “Local”. If key “Scheduled” is **not** present, the existence of “GameObject” can be guaranteed.

Pre-processing the info Dictionary

When you are using the OnNotification() method to process events, you can pre-inspect and pre-process the info dictionary that will eventually be passed to your notification handling method.

All you need to do is to override

```
public virtual void preProcessInfo(Dictionary<string, object> theInfo)
```

in your code, and that preProcessInfo method will be invoked immediately, before any filtering is done.

Posting (Sending) Notifications

Note that a script does not have to be signed up for notifications in order to be able to send one. SIP implements sendNotification and postNotification messages – their only difference is their spelling (postXXX vs. sendXXX) – this is done simply for convenience.

postNotification(name) Variants

```
void postNotification(string theNotificationname)
void sendNotification(string theNotificationName)
```

Sends out a notification with name theNotificationName. The "Event" field is automatically filled with the string "cfxGenericNotification".

```
void postNotification(string theNotificationname, float delay)
void sendNotification(string theNotificationName, float delay)
```

As above, except that the notification is sent after delay seconds. The notification is scheduled centrally on the notification manager.

```
public void postNotification(string theNotificationName,
float delay, bool local)
public void sendNotification(string theNotificationName,
float delay, bool local)
```

As above, but also allows you to optionally schedule the notification locally on the script that is sending the notification. The notification is scheduled locally if you pass "true" for local, otherwise the notification is scheduled centrally on the notification manager.

postNotification(name, event) Variants

```
void postNotification(string theNotificationName, string theEvent)
void sendNotification(string theNotificationName, string theEvent)
```

Sends out a notification with name theNotificationName, and the "Event" field set to theEvent.

```
void postNotification(string theNotificationName, string theEvent,
float delay)
void sendNotification(string theNotificationName, string theEvent,
float delay)
```

As above, except that the notification is sent after delay seconds. The notification is scheduled centrally on the notification manager.

```
public void postNotification(string theNotificationName,
string theEvent, float delay, bool local)
public void sendNotification(string theNotificationName,
string theEvent, float delay, bool local)
```

As above, but also allows you to optionally schedule the notification locally on the script that is sending the notification. The notification is scheduled locally if you pass “true” for local, otherwise the notification is scheduled centrally on the notification manager.

postNotification(name, GameObject) Variants

```
public void sendNotification(string theNotificationName,
GameObject theObject)
public void postNotification(string theNotificationName,
GameObject theObject)
```

Sends out a notification with name theNotificationName, and “TheObject” field set to the value of theObject

```
public void sendNotification(string theNotificationName, GameObject
theObject, float delay)
public void postNotification(string theNotificationName, GameObject
theObject, float delay)
```

As above, except that the notification is sent after delay seconds. The notification is scheduled centrally on the notification manager.

```
public void sendNotification(string theNotificationName,
GameObject theObject, float delay, bool local)
public void postNotification(string theNotificationName,
GameObject theObject, float delay, bool local)
```

As above, but also allows you to optionally schedule the notification locally on the script that is sending the notification. The notification is scheduled locally if you pass “true” for local, otherwise the notification is scheduled centrally on the notification manager.

postNotification(name, theDict) Variants

```
void postNotification(string theNotificationName, Dictionary<string,
object> theDict)
void sendNotification(string theNotificationName, Dictionary<string,
object> theDict)
```

Sends out a notification with the name theNotificationName and all the information contained in theDict. If theDict does not contain an “Event” entry, one is added containing the value “cfxGenericNotification”.

```
void postNotification(string theNotificationName,
Dictionary<string, object> theDict, float delay)
void sendNotification(string theNotificationName,
Dictionary<string, object> theDict, float delay)
```

As above, except that the notification is sent after delay seconds. The notification is scheduled centrally on the notification manager.

```
public void postNotification(string theNotificationName,
Dictionary<string, object> theDict, float delay, bool local)
public void sendNotification(string theNotificationName,
Dictionary<string, object> theDict, float delay, bool local)
```
As above, but also allows you to optionally schedule the notification locally on the script that is sending the notification. The notification is scheduled locally if you pass “true” for local, otherwise the notification is scheduled centrally on the notification manager.

postNotification(name, event, theDict) Variants

```
void postNotification(string theNotificationName, string theEvent,
Dictionary<string, object> theDict)
void sendNotification(string theNotificationName, string theEvent,
Dictionary<string, object> theDict)
```
As above, but also automatically copies the contents of theEvent into the “Event” field in theDict.

```
void postNotification(string theNotificationName, string theEvent,
Dictionary<string, object> theDict, float delay)
void sendNotification(string theNotificationName, string theEvent,
Dictionary<string, object> theDict, float delay)
```
As above, except that the notification is sent after delay seconds. The notification is scheduled centrally on the notification manager.

```
public void postNotification(string theNotificationName,
string theEvent, Dictionary<string, object> theDict, float delay,
bool local)
public void sendNotification(string theNotificationName,
string theEvent, Dictionary<string, object> theDict, float delay,
bool local)
```
As above, but also allows you to optionally schedule the notification locally on the script that is sending the notification. The notification is scheduled locally if you pass “true” for local, otherwise the notification is scheduled centrally on the notification manager.

Queries

Queries work similar to Notifications, with the added benefit that the queried objects can return a result. Queries are also voluntary co-op, so objects that want to respond to a query must sign up and can resign any time. If you sign up to answer queries, you must implement an OnQuery method that is invoked when the query arrives.

Queries only work is there is a cfxNotificationManager prefab in your scene.

Like notifications, queries are always named, meaning that to sign up for a query, you sign up to be queried for a specific query name, e.g. "HasTag". Unlike notifications, a script can elect to be asked every time a query comes in, by providing a wildcard "*" (single asterisk) as subscribed notification name.

Queries can be invoked with, and can return a scope (i.e. a set of responders to send a query to or a set of responders that have answered to a query). You can't create a scope by yourself, they are always acquired and refined with a runQuery invocation.

!	Query scopes are volatile – they do not persist through scene changes; nor can they be saved and later successfully retrieved from storage.

Signing up to answer queries

```
void respondToQueryNamed(string queryName)
```
Subscribes to the query named queryName. You can provide the Wildcard "*" if you want your script to be invoked for every query, or use the respondToAllQueries convenience method described below.

!	You must not invoke this method during OnQuery(). You can freely invoke this method during OnNotification().

```
void respondToAllQueries()
```
Subscribes your script to all queries. Note that if your script subscribes to all queries and additionally subscribes to a specific query name, your script will be called twice for that query name: once for the wildcard, once for the specific entry. It's up to you to ensure that this resolves well (hint: don't mix both modes).

!	You must not invoke this method during OnQuery(). You can freely invoke this method during OnNotification().

Withdrawing from answering queries

```
void withdrawAllQueryResponses()
```
Withdraws (cancels) all subscribed queries for your script up to this point.

!	You must not invoke this method during OnQuery(). You can freely invoke this method during OnNotification().

```
void withdrawQueryResponseFor (string queryName)
```
Withdraws (cancels) your script's subscription to answer the query name queryName. If your script is not subscribed to this particular queryName, the withdraw request is ignored.

!	You must not invoke this method during OnQuery(). You can freely invoke this method during OnNotification().

OnQuery Variants

Your query handling methods must provide a response object, or return NULL to indicate that they do not have an answer. In that case, the script will not be added to the query's responder scope. If you return anything but NULL, the script is added to the responder scope.

Note that all OnQuery implementations must have the **override** keyword.

```
public override object OnQuery(string queryName)
```
Invoked if you signed up to a query with name queryName.

```
public override object OnQuery(string queryName, string lookFor)
```
Invoked with queryName and the "LookFor" entry in the query info copied to lookFor.

!	If the data dictionary provided with runQuery() does not contain a "LookFor" entry, this method will not be invoked.

```
public override object OnQuery(string queryName,
Dictionary<string, object> info)
```
Invoked with query name and the info Dictionary that contains all query details.

```
public override object OnQuery (string queryName, string lookFor,
Dictionary <string, object> info)
```
Like above, except that lookFor is prefilled from info's "LookFor" entry.

!	If the data dictionary provided with runQuery() does not contain a "LookFor" entry, this method will not be invoked.

Implementing your own OnQuery

```
public void respondToQueryNamed(string queryName,
cfxNotificationManager.queryHandler handler)
```

If you want to have your own method invoked for a query rather than using OnQuery, you need to sign up by using above respondToQueryNamed() variant. To be able to pass a method, it must match the following pattern:

```
object myQueryResponse(Dictionary<string, object> info)
```

Similar to notifications, the info Dictionary contains all supplementary information. Of special important are the “Query” and “LookFor” entries.

If your method has no answer, it must return NULL. In that case, the script will not be added to the query’s responder scope. If you return anything but NULL, the script is added to the responder scope.

Running a Query

When you run a query, the Query manager sends your query to all scripts that have signed up to answer a query with the same name as the one you provided. If you provide a scope with the query, the Manager restricts the query to all scripts that have subscribed to the query name and are on the scope list.

These scripts then decide if they return a value. The Query Manager compiles all answers and finally returns a Dictionary of all responders and their answers.
SIP takes this information, breaks it down into the answers (what the responders returned) and scope (who responded), and makes this information accessible separately, depending on which runQuery variant you choose.

runQuery(queryName)

```
List<object> runQuery(string queryName)
```

Runs the query named queryName and returns all responses as a list.

```
List<object> runQuery(string queryName, List<string> scope)
```

As above, but restricts the query to the responders listed in scope. To acquire a scope, use a variant of runQuery that returns a scope. You can provide null for scope. In that case it is disregarded.

```
List<object> runQuery (string queryName,
out List<string> responders)
```

Runs thequery and **returns a scope:** a list of all responders that have returned a non-NULL object in responders. Note that the scope encodes volatile Query Manager information that is unusable otherwise. You can use the same variable for both scope and responders to rapidly narrow your search.

```
List<object> runQuery(string queryName, List<string> scope,
out List<string> responders)
```

Runs the query with a scope and also **returns a scope.**

runQuery(queryName, lookFor)

```
List<object> runQuery(string queryName, string lookFor)
```

Runs the query named queryName with the additional parameter “LookFor” set to the value lookFor.

```
List<object> runQuery(string queryName, string lookFor,
List<string> scope)
```

As above, but restricts the query to the responders listed in scope. To acquire a scope, use a variant of runQuery that returns a scope. You can provide null for scope. In that case it is disregarded.

```
List<object> runQuery(string queryName, string lookFor, out List<str
ing> responders)
```

Runs the query and **returns a scope.**

```
List <object> runQuery(string queryName, string lookFor,
List<string> scope, out List<string> responders)
```

Runs the query with scope and also **returns a scope.** You can use the same variable for both scope and responders to rapidly narrow your search.

runQuery(queryName, info)

```
List<object> runQuery(string queryName,
Dictionary<string, object> info)
```

Runs the query named queryName and makes the contents of the dictionary info available to all responders.

```
List<object> runQuery(string queryName,
Dictionary<string, object> info, List<string> scope)
```

As above, but restricts the query to the responders listed in scope. To acquire a scope, use a variant of runQuery that returns a scope. You can provide null for scope. In that case it is disregarded.

```
List<object> runQuery(string queryName, Dictionary<string,
object> info, out List<string> responders)
```

Runs the query and **returns a scope**.

```
List<object> runQuery(string queryName,
Dictionary<string,object> info, List<string> scope,
out List<string> responders)
```

Runs the query with a scope and **returns a scope.** You can use the same variable for both scope and responders to rapidly narrow your search.

Helper Methods

SIP also implements a number of helper methods for you that makes using notifications or queries even more comfortable. These methods are available as soon as you inherit from cfxNotificationBehaviour (SIP's root class) or any of its descendants.

Query result processing

convertToGameObjects

```
public List<GameObject> convertToGameObjects(List<object> inList)
public List<GameObject> convertToGameObjects(List<object> inList,
bool sorted)
```

Use this method if your query responders return a list of GameObjects. The methods converts the List of objects into a List of GameObjects, and can optionally sort the whole list by the Name property (ascending).

Miscellaneous

Sorting GameObjects

```
public List<GameObject> sipSortGameObjectList(List<GameObject>
unsorted)
```

Use this method to sort a list of GameObjects by name. This method is provided simply because it's a common request and comes in handy when dealing with Queries (note that the convertToGameObjects method provides built-in functionality for that)

Info Dictionary: retrieving values

To access the keys in the info dictionary, SIP provides a number of convenience methods that all conform to our recommended best practice coding style.

!	We strongly recommend you use these accessor methods, for the following reasons: • they conform to best practice coding style and catch possible null exceptions reliably. • they support strong error logging. If something seems wrong in your script, simply check the 'verbose' box for the script, and check the log to ensure that all store and retrieve methods work as intended.

Some variants of these methods use an implicit dictionary for coding convenience. This is only guaranteed to be set when you use the standard OnNotification() method to process incoming notifications.

Retrieving a generic object
```
public object fetchObject(Dictionary<string, object> info,
string theKey)
public object fetchObject(Dictionary<string, object> info,
string theKey, object defaultValue)
```
Retrieves the object stored under theKey from the info dictionary. If the key does not exist, or anything else goes wrong, the method returns defaultValue as a generic object (or null if defaultValue is not specified).

Implicit versions that are only guaranteed to work inside OnNotification():
```
public object fetchObject(string theKey)
public object fetchObject(string theKey, object defaultObject)
```
As above, except the info dictionary is inferred from the OnNotification notification handler. If you supplied your own handler, you should not use these methods unless you set theImplicitInfo first.

```
public string fetchString(Dictionary<string, object> info,
string theKey)
public string fetchString(Dictionary<string, object> info,
string theKey, string defaultValue)
```
Retrieves the string stored under theKey from the info dictionary. If the key does not exist, or anything else goes wrong, the method returns defaultValue as a string (or null if defaultValue is not specified).

Implicit versions that are only guaranteed to work inside OnNotification():
```
public string fetchString(string theKey)
public string fetchString(string theKey, string defaultObject)
```
As above, except the info dictionary is inferred from the OnNotification notification handler. If you supplied your own handler, you should not use these methods unless you set theImplicitInfo first.

```
public GameObject fetchGameObject(Dictionary<string, object> info,
string theKey)
```

```
public GameObject fetchGameObject(Dictionary<string, object> info,
string theKey, GameObject defaultValue)
```
Retrieves the GameObject stored under theKey from the info dictionary. If the key does not exist, or anything else goes wrong, the method returns defaultValue as a string (or null if defaultValue is not specified).

Implicit versions that are only guaranteed to work inside OnNotification():
```
public GameObject fetchGameObject(string theKey)
public GameObject fetchGameObject(string theKey,
GameObject defaultObject)
```
As above, except the info dictionary is inferred from the OnNotification notification handler. If you supplied your own handler, you should not use these methods unless you set theImplicitInfo first.

```
public float fetchFloat(Dictionary<string, object> info, string theKey)
public float fetchFloat(Dictionary<string, object> info,
string theKey, float defaultValue)
```
Retrieves the float value stored under theKey from the info dictionary. If the key does not exist, or anything else goes wrong, the method returns defaultValue as a float (or 0 (float) if defaultValue is not specified).

! SIP passes floats as strings. You must either add the float converted to string yourself, or use the convenience function addFloatToInfo (see below). You must not try to add a float to the info dictionary.

Implicit versions that are only guaranteed to work inside OnNotification():
```
public float fetchFloat(string theKey)
public float fetchFloat(string theKey, float defaultValue)
```
As above, except the info dictionary is inferred from the OnNotification notification handler. If you supplied your own handler, you should not use these methods unless you set theImplicitInfo first.

```
public int fetchInt(Dictionary<string, object> info, string theKey)
public int fetchInt(Dictionary<string, object> info, string theKey,
int defaultValue)
```
Retrieves the integer value stored under theKey from the info dictionary. If the key does not exist, or anything else goes wrong, the method returns defaultValue as a integer (or 0 (integer) if defaultValue is not specified).

! SIP passes integers as strings. You must either add the integer converted to string yourself, or use the convenience function addIntToInfo (see below). You must not try to add an integer to the info dictionary.

Implicit versions that are only guaranteed to work inside OnNotification():
```
public int fetchInt(string theKey)
public int fetchInt(string theKey, int defaultValue)
```
As above, except the info dictionary is inferred from the OnNotification notification handler. If you supplied your own handler, you should not use these methods unless you set theImplicitInfo first.

```
public Color fetchColor(Dictionary<string, object> info, string theKey)
public Color fetchColor(Dictionary<string, object> info, string theKey,
Color defaultValue)
```
Retrieves the Color stored under theKey from the info dictionary. If the key does not exist, or anything else goes wrong, the method returns defaultValue as a Color (or Color.black if defaultValue is not specified).

! SIP passes Colors as HTML-encoded strings. You must either add the Color converted to string (via ColorUtility, remember to prepend the hash!) yourself, or use the convenience function addColorToInfo (see below). You must not try to add a color to the info dictionary.

Implicit versions that are only guaranteed to work inside OnNotification():
```
public Color fetchColor(string theKey)
public Color fetchColor(string theKey, Color defaultValue)
```
As above, except the info dictionary is inferred from the OnNotification notification handler. If you supplied your own handler, you should not use these methods unless you set theImplicitInfo first.

```
public bool fetchBool(Dictionary<string, object> info, string theKey)
public bool fetchBool(Dictionary<string, object> info, string theKey,
bool defaultValue)
```
Retrieves the boolean value stored under theKey from the info dictionary. If the key does not exist, or anything else goes wrong, the method returns defaultValue as a Bool (or false if defaultValue is not specified).

! SIP passes Boolean values as strings with the possible values of “true” or “false” (more precisely: any value other than the string “true” (case-insensitive) is interpreted as false by the fetch method. Use the accessor AddBoolToInfo to ensure compatibility.

Implicit versions that are only guaranteed to work inside OnNotification():
```
public bool fetchBool(string theKey)
public bool fetchBool(string theKey, bool defaultValue)
```
As above, except the info dictionary is inferred from the OnNotification notification handler. If you supplied your own handler, you should not use these methods unless you set theImplicitInfo first.

```
public Vector3 fetchVector3(Dictionary<string, object> info,
string theKey)
public Vector3 fetchVector3(Dictionary<string, object> info,
string theKey, Vector3 defaultValue)
```
Retrieves the Vector3 value stored under theKey from the info dictionary. If the key does not exist, or anything else goes wrong, the method returns defaultValue as a Vector3 (or Vector3.zero if defaultValue is not specified).

! Vectors are stored as three separate strings in info, with keys generated from theKey by appending '.VectorX', '.VectorY' and '.VectorZ'. Use the accessor AddVector3ToInfo to ensure compatibility.

Implicit versions that are only guaranteed to work inside OnNotification():
```
public Vector3 fetchVector3(string theKey)
public Vector3 fetchVector3(string theKey, Vector3 defaultValue)
```
As above, except the info dictionary is inferred from the OnNotification notification handler. If you supplied your own handler, you should not use these methods unless you set theImplicitInfo first.

Info Dictionary: adding/storing values

```
public void addFloatToInfo(Dictionary<string, object> info,
string theKey, float theFloat)
```
Adds the float theFloat, converted to string, under the key theKey to the dictionary info.

```
public void addIntToInfo(Dictionary<string, object> info,
string theKey, int theInt)
```
Adds the integer theInt, converted to string, under the key theKey to the dictionary info.

```
public void addBoolToInfo(Dictionary<string, object> info,
string theKey, bool theBool)
```
Adds the boolean theBool, converted to string, under the key theKey to the dictionary info. The String is “TRUE” if theBool is true. Any other string is interpreted as false.

```
public void addColorToInfo(Dictionary<string, object> info,
string theKey, Color theColor)
```
Adds the Color theColor, converted to a HTML string, under the key theKey to the dictionary info.

```
public void addStringToInfo(Dictionary<string, object> info,
string theKey, string theString)
```
Adds the string theString as value under the key theKey to the dictionary info. The same can be achieved by

```
      info [theKey] = theString;
```

except that this method adds null checking for all variables (and strong logging if verbose is enabled).

```
public void addGameObjectToInfo(Dictionary<string, object> info,
string theKey, GameObject theGameObject)
```
Adds the GameObject theGameObject under the key theKey to the dictionary info. The same can be achieved by

```
      info [theKey] = theGameObject;
```

except that this method adds null checking for all variables (and strong logging if verbose is enabled).

```
public void addVector3ToInfo(Dictionary<string, object> info,
string theKey, Vector3 theVector)
```
Adds the Vector3 theVector as three separate strings under the base key theKey to the dictionary info.

Suspend / Resume Notifications

Suspend the Notification Manager

`public void suspendAllNotifications()`

Suspends all notifications. Any notifications that are posted (by any script) while the notifications are suspended, are put into a queue. Scheduled notifications continue to count down normally. Should their timer expire while notifications are suspended, the notification is put in the queue.
The notifications are posted when notifications are resumed. Note that you can filter the queue before resuming (see below).

`public bool notificationsAreSuspended()`

Returns true if notifications are suspended.

`public List<string> pendingNotificationNames()`

Returns a list of all currently ending notification names, in the order they first appeared in the queue. If there are two pending notifications of the same name, only the first is listed. You can use this list to filter the queue before resuming notifications (see below)

`public List<string> pendingNotificationNames(bool unique)`

As above, but if you pass 'false' to unique, the list contains all notification names in the order they are in the queue, including duplicates.

Resume Notifications

`public void resumeNotifications()`

Resumes sending out notifications. All notifications that have been put in the queue are sent out in the order they were put in the queue. If a scheduled notification was put in the queue, it is scheduled now, with the original delay.

`public void resumeNotifications(bool discardAll)`

As above, but passing true with discardAll will remove all notifications from the queue, resuming notifications without posting any notification that were queued during suspension.

`public void resumeNotifications(string discardNotificationNamed)`

Resumes sending out notifications. All notification names in the queue that match the string discardNotificationsNamed are removed. All remaining notifications that have been put in the queue are sent out in the order they were put in the queue. If a scheduled notification was put in the queue, it is scheduled now, with the original delay.

`public void resumeNotifications(List<string> discardNotificationsNamed)`

As above, except that all notifications in the queue that match any of the strings in discardNotificationsNamed are removed.

Other / Miscellaneous

Unique Notification Names

```
public string UniqueNotificationName(string inNotificationName)
```

This method mill return a new string based on the script's instance in the scene. It will always return the same result based on inNotificationName within the same instance, and *only* inside the same instance. Two different instances of the script will always return different results *across* instances. This allows you to create instance-specific notification names, and then use multiple copies of the script in your scene without them triggering each other.

Part IX: Geek Sector A

Here be Dragons!

This is the part that, if you read it, officially makes you a geek. Here we'll delve into more advanced topics like providing your own OnNotification variants and even multiple notification handlers in the same script. And we'll look at how you pass structured information with your notifications. And we'll drop pretending that you are a Beginner. If you need this stuff, you know you aren't. So, buckle up, you're in for a great ride!

Providing your own context

You can add a lot more information to a notification than just the fact that something has happened (the doorbell ringing / the notification name) and a simple qualifier (the event name). But those two pieces of information will usually suffice to get you wherever you want to go. What follows below will only serve get you there in style.

! SIP provides convenience methods to add, and retrieve, GameObjects, Vector3, bool, float, integer and color values to and from the info dictionary. You should use those methods instead whenever possible, as they are coded using best practice and guarantee compatibility with future versions

Once you get more familiar with notifications you may wish to pass along some more information: which enemy was sighted, where was it sighted, where was it heading, etc. This context information you can pass along with the notification itself, and the receiving scripts can access that information – provided they know that it is there, and how to interpret it. All supplementary information is provided in the form of a Dictionary.

Some information is included by default (as long as you use the convenience interface, of course), for example who (which GameObject) sent out the notification. To access this information, simply request the value for the key, e.g.

```
GameObject theSender = info["GameObject"] as GameObject;
```

Information that is always provided with every notification is:
"GameObject" : the game object that is posting the notification
"Time": the Unity Time value (as a string) since start of the session.
“Scheduled” – if this key is present, the notification was scheduled.

You can add your own information to any notification. All you do is create a Dictionary<string, object> and add your information that you identify with your own keys. For example, to add the detected enemy's GameObject to the info you are passing along with the notification, you could do the following:

```
Dictionary<string, object> myInfo = new Dictionary<string, object>();
myInfo.Add("Enemy", theEnemy);
```

The notified scripts can then access this information using the same key. Note that this requires that both scripts know that there is an enemy passed along with the information, and that this is stored under the "Enemy" key.

```
GameObject detectedEnemy = info["Enemy"] as GameObject;
```

Reserved key for the info dictionary

Unlike with notification names, there are a few keys that you can't (or rather: should not) use to pass information, because the notification manager fills this in automatically, possibly overwriting whatever you have stored there. These are:

- "GameObject"
- "Time"
- "Event" (unless intentionally)
- “Scheduled” (NEVER screw with this key!)

Special care must be made when you add floats or integer numbers (or colors) to the information dictionary. We recommend that you convert them to strings prior to sending, and convert them back once received like so:

```
Info["Damage"] = "250.0";
Info["Damage"] = theFloat.ToString();
```

and

```
float damage = 100.0f; // set default
// guarded conversion
if (Info.ContainsKey("Damage"))
      damage = float.Parse(Info["Damage"] as string);
```

Always precede your info access with a guard. This protects your script against errors or unintended side effects.

But just how do you pass, and retrieve, that information? Well, as we have indicated before, SIP implements a number of convenience functions for you, and sending as well as receiving notifications supports multiple "signatures" or "variants" (or what ever you call an "overloaded" function call). Some of them include passing the dictionary, e.g.

```
Dictionary<string, object> Info = new Dictionary<string, object>()
```

The same is true for OnNotification(). There are multiple possible implementation variants, and you can pick any one (as long as it is exactly one), e.g.

```
OnNotification(string notificationName, Dictionary<string, object>
Info)
```

! Even though not accessible in some variants, the info dictionary is *always* sent with a notification.

Selective Notifications: Multiple Responders

Of course, it sometimes may be advantageous to get a more finely-grained control over notification response. Up until now, we have explored SIP's convenience interface, which should get you started, and enable you to significantly simplify your code. That being said, SIP's central OnNotification() mechanism can become a bottleneck similar to Update(). This is because all Notifications that you subscribeTo are routed to OnUpdate, and given enough subscriptions, your OnNotification method will resemble the ugly if-then-else-return 'shunting hell' that Update was.

Luckily, that is easy to overcome. Much of SIP's convenience was engineered in light of Unity's special requirements. It was one of our goals that although SIP must be easy to use, we would not compromise flexibility. So as soon as your OnNotification method start to look crowded, you can opt to use SIP's more powerful, but ever so slightly less convenient, 'direct subscription' method.

```
subscribeTo(string theNotificationName, myResponderMethod)
```

Where myResponderMethod is the method that should be invoked when a subscribed notification is received

As you can see, subscribing your own responder is quite similar to the built-in code. Being slightly lower-level, the subscription method eschews some of the comfort of its higher siblings:

- No Filtering. SIP does not provide filtering for any notification you subscribe to via this method.

- Fixed, enforced Method Parameters. Since you supply the method that is called upon notification, its parameters must conform to

  (string, Dictionary<string, object>)

This means that any method that fits that description is OK, e.g.
- myMethod(string something, Dictionary<string, object> myVar)
- callMeMaybe(string foo, Dictionary<string, object> bar)

but not

- thisIsWrong(string something) -- missing Dictionary <string, object>, nor
- notRightEither(string anotherThing, Dictionary<string, string>) – wrong type of Dictionary, nor
- wrongAgain(int a) – completely wrong parameter list
- tooManyParams(string s, Dictionary<string, object>, int tooMuch) – the int parameter

Additionally, the method must be implemented in the same script. If it doesn't, you'll receive a compiler error and can only proceed once you have added such a method.

That way, you can easily subscribe to multiple notifications, and provide your own methods that are invoked when the correct notification arrives. You can assign the same method to different notification names, and you can mix this way of subscribing to notifications with the more convenient one – which is a common design pattern: subscribe to the routine notifications through the convenience methods and OnNotification, and prove a highly specialized method for one or two exceptional notifications.

Advanced Topics

Improving performance

Using a central notification manager, and an notification-based event model, can significantly improve your code readability and performance. During execution time, one of the costliest calls are your scripts 'Update()' messages. Since most scripts that use the Update call use it to monitor or look for a specific change, most Update calls are spent only for event management, something that should be, but due to the acute lack of notification management in Unity isn't, centralized. This means that instead of one script that runs Update and generates all necessary events, you have tens, if not more than one hundred scrips that run Update, wasting lots of resources.

Most scripts can be taken apart (modularized) to focus on a single task that is triggered by an event (notification). This task possibly turns on Update() or triggers other events (notifications) that are distributed over your scene in different scripts. This approach is not only much more flexible, it also encapsulates functionality, making your code more resilient against mishaps (how many times did you change a single line of code just to mysteriously have you entire game fail) and easier to maintain. It also can make it perform better because only those scripts execute that need to become active.

Query-enabled generic scripting

One powerful use of the query manager is that you can use it as a central data repository that servers your script scene-specific information. This allows you to write generic code that implements the general game logic without including scene-specific code. With this approach you usually use level-specific prefabs that implement the details that you can switch between levels while the underlying code remains the same.

For example, different wizards each have a set of spells that essentially worked the same (e.g. guided projectile, unguided projectile, aoe attack) but have radically different visuals. Or the armies of the various factions in an RTS have vehicles that all have similar classes (Heavy, Light, Artillery), but look and move differently.

To implement this, we use objects that can be queried for the correct unit prefab. In the RTS game, we create a spawn manager for each faction that only responds when the correct faction is requested, and then responds with the prefab for the requested unit. Level design is simplified: we now can write generic unit handling code, and then add and remove prefab spawner objects for each faction only when the level begins (or even later, when new players drop in). That way a player can choose different factions and different numbers of enemies when they start the level. This design is so successful because SIP is designed from ground up with strong support to handle unspecified numbers of objects – as opposed to having the number of objects parametrized (in variables) or (shudder) hard-coded.

For example, to request a new tank prefab (which is the heavy class) for the "Red" faction, the following line could be used:

```
GameObject newVehicle = runQuery("getUnitPrefab",  "Red", "Heavy");
```

Above queries the scene for the prefab for Red faction's Tank. Only the spawn manager for Red answered with the correct prefab.

```
public override object OnQuery(string queryName, string lookFor,
Dictionary<string, object> info) {
    if (lookFor != "Red") return null;
    // info key "Unit" contains unit we look for
    string unitRequested = info["Unit"];
    // unitPrefabs contains the prefab for each unit
    return unitPrefabs[unitRequested];
}
```

Since these spawn managers could be added and removed freely, one of the first things the scene logic did was ask how many factions are present by executing a roll call:

```
List<string> factions = runQuery ("factionName");
```

To which all prefab spawners respond with their faction name.

For the wizard, we implemented something very similar, except that there was only one wizard 'spell' book that was defined by each level. Getting the correct spell was now merely an enquiry to the scene

```
GameObject theSpell = runQuery("getSpellPrefab", "aoe_attack");
```

And the 'spellbook' object in the scene was simply the only object that signed up to answer this question

```
public override object OnQuery(string queryName, string lookFor) {
    return spellPrefabs[lookFor];
}
```

! Note that in above examples the patterns for runQuery don’t match SIP, which returns a List<object>, not GameObject. We implemented overloaded methods of runQuery for this purpose to return the first item in the array as GameObject for this.

They are available in clean form as “runQueryAndFetchFirst” as a bonus for those who have read this documentation up to this point.

Best Practices

There are some tips that may help you avoid common pitfalls with SIP

Always guard your response

While it is annoying to enclose your response code in an ‘if’ clause even if you know that you have subscribed to only a single notification or query, it is much better to make sure. Because you know that it will not remain that way

```
if (notificationName == "Dissolve") {
        ...
}
```

Always return inside your response

Unless there is a good reason not to (e.g. common handling code at the end), always return at the end of your guarded response

```
if (notificationName == "Dissolve") {
        ...
        return;
}
```

Always Debug.Log when code reaches the end of your response

When you always guard and exit, you should never reach the end of your responder method. Make sure you log if you do, because this makes it very easy to spot an unhandled notification

```
public override void OnNotification (string notificationName) {
    if (fading)
        return;

    if (notificationName == "Dissolve") {
        ...
        return;
        }

    if (notificationName == "Resolve") {
        ...
        return
    }
    Debug.Log("OnNotification not handled for: " + notificationName);
}
```

Always guard info access

Accessing the info dictionary directly is dangerous. You should not trust that the invoking script remembered to fill all the relevant keys and provide defaults. That way your code is not only safe, it will also not throw exceptions when you try to access information that isn’t there.

```
string playerName = "Not Named";
if (info.ContainsKey ("PlayerName")) playerName = info["PlayerName"];
```

info uses string to store floats and colors

Remember that the info dictionary stores floats as strings. Use ToString to store and float.Parse to retrieve floats:

```
info["theFloat"] = f.ToString ();
```

and

```
float f = 1.0f;
if (info.ContainsKey ("theFloat")) f = float.Parse(info["theFloat"]);
```

! Use the Helper Methods fetchFloat and addFloatToInfo, fetchVector3 and addVector3ToInfo, as well as fetchColor and addColorToInfo to correctly pass and retrieve floats and colors to the info dictionary. Those methods are guaranteed to comply with best practice.

Integration with PlayMaker

[Note: Still in development, moved to a future point release]

"PlayMaker" from Hutong Games is a well-executed, successful visual scripting extension for unity. Using a visual metaphor and a concept called 'Finite State Machine', it makes designing programs accessible to people who, despite evidence to the contrary, believe they can't program. It allows people overcome their fear of letters and semicolons by hiding them behind boxes and arrows. It has proven beyond doubt that one of the greatest obstacles to programming isn't ability, but perception. Playmaker provides an important service, as it enables artist to give form to their thoughts in a way that a machine can use to structure the flow of control. It may well be one of the most important enablers of talent in Unity's universe.

If you look closely at PlayMaker, you'll quickly realize that below its interface sits a very large notification machine – almost everything in PlayMaker is controlled by generating, and consuming, notifications (called "Triggers" or "Events"). Since Playmaker is so prominent in Unity, it's a good idea to provide an easy mechanism to interface SIP with PlayMaker. There is a significant difference between PM and SIP, and they can be used to complement each other nicely: PM's FSM focuses mainly on managing an object's state based on clearly defined rules (state transitions) that are triggered by events (what in SIP are notificatons). SIP focuses on allowing scripts to pass messages to other scripts, usually when something interesting happens. PM's FSM are object-focused, with mostly local triggers and support for global triggers; SIP always works on a centralized global level, while local support is incidental (a script can receive a notification it sends itself). Merging the two can be highly beneficial, and using the provided prefabs, it's also quite easy.

We provide two levels of PM integration.

PM Bridge Listener
The first one is a simple bridge: it listens to a notification, and then triggers a named event for the PM FSM. Since the notification it listens for and the PM event it triggers can be specified separately, the bridge can also simultaneously translate SIP notifications into PM.

The second prefab is somewhat more involved. It can set property values of an FSM before it triggers an event, and relies on your script to send the relevant data via the info dictionary. It requires that you tell the Prefab each variable name and type that you want to set or get. For example, if the FSM uses a variable of type integer that is named 'HighScore', this is entered as such in the interface translation dictionary of the prefab. When you notification arrives, the prefab scans the info dictionary for any variables listed in the translation dictionary, and if present, sets the variable according to the value passed in the notification's info dictionary. Please be advised: there is a lot that can go wrong here: you need to enter both name and type of the variable perfectly each time –in PM, in your calling script, and in the Prefab. Since the pay-off can be tremendous, it's worth doing it – just make sure that you are testing sufficiently before deployment

Getting variables is implemented via Query, and returns all values

Integration works via bridge prefabs that you attach to the objects that contain PM FSM.

The Future

All software evolves. SIP is useful, but still can improve. There are a number of ways we are looking into making the package more useful to you, and we have outlined some of them below. Then there are many others we didn't think of – but perhaps you just did. If you have found an elegant way that involves SIP and want to share, or are looking at an issue you think should be more easy to solve – let us know. We love hearing from you, and promise to get back to you as soon as we can.

Future expansions

- Script-to-script (asynchronous) messaging.
- Integration with specific assets

Planned, but currently out of scope

- Background capable queries – requires Unity's API to become thread-safe

Your Own Ideas, Comments, Requests

SIP is still in its first major release cycle. It has been used in the development of only a few titles, and many features can be either enhanced, amended, or made more accessible. There are many Prefabs you might want but we haven't thought of. Maybe you need some hints on how to do something, maybe you want to share your own creation. In any way: contact us. We can be reached at

support@cfxsoftware.com if you have a question

or

info@cfxsoftware.com

for any other purpose.

Acknowledgements

Imagery

Digital Images: cf/x stock photography
Digital image processing : cf/x alpha, cf/x photo
Artwork: cf/x; cf/x logo designed by Iris Schwarz

Special Thanks

Special thanks go to our alpha, beta and gamma testers, proofreaders, models, legal advice, vanity support, and everyone else helping us.