Software Installation:

The main software repository is at https://github.com/organizations/dopplerimager. There are a number of repositories here, one for each instrument (containing instrument-specific code), one for the core SDI control software (`/sdi`) and one for the code to build the dll wrapper (`/sdiexternal`). This description will assume that the instrument-specific code for the new instrument has already been uploaded to GitHub (and exists as for example, `/poker`).

In `/sdi` there is an install script called install.bat. An easy way to install the software on a new machine is to go to https://github.com/dopplerimager/sdi, click on `install.bat`, and click on RAW (this will download the file). Open up a command prompt (on Windows 7 and higher you will need to right-click on the command prompt icon and select "Run As Administrator"). Change to the directory where `install.bat` was downloaded. As an example, I'll assume you want to install both the core SDI software and the Poker Flat instrument-specific software. At the command prompt, you would type:

```
install.bat sdi poker
```

This will do a couple of things: it will first try to create the directory c:\users\sdi3000. If it is unable to do so, it will terminate. Assuming it succeeded, it will change to that directory, and run the Git commands to download (clone) the respective repositories from GitHub which were passed as arguments to the bat file (sdi and poker). If these clone's succeed, the script will create the default directory tree (directories for data, log, settings, phasemaps, screencaptures). It will then print a friendly message reminding you what to do next (updating IDL and windows search paths, etc.).

If either the `sdi` or `poker` repositories already existed on the local machine, the script will terminate and ask you to remove them. Assuming that what you actually want to do is update them, and not re-install, then see below.

Note: Once installed and running, you should open up and actually use each of the main plugins (StepsPerOrder, Phasemapper, Spectrum) manually at least once in order to provide those plugins with correct settings for auto operation.

Using Git:

<u>Update local software from GitHub:</u>
In order to update an installation on a local machine, incorporating any new changes that might be in the GitHub repository, do the following (I'll assume you want to update the core SDI repo):

1. Open git bash (this is a command line, bash-like interface)
2. `cd c:/users/sdi3000/sdi` (note the forward slashes)
3. `git pull`

Assuming all went well, this will download any changes to any files, and incorporate them into the local repo.

<u>Update GitHub repo from local software:</u>

If you make changes to a local copy of a repository, you should send those updates back to the GitHub repository so things don't get out of sync. To do this (again assuming you have made a change to the core SDI software):

1. Open git bash
2. `cd c:/users/sdi3000/sdi`
3. `git push`
4. you will need to enter username and password

This will update the online repo with your local changes.

Creating a new GitHub repository:
You might want to do this for example if you write instrument-specific code for a new instrument.

1. Go to https://github.com/organizations/dopplerimager
2. On the right there should be a button labeled "New Repository", click on this
3. Enter a name and optional description, leave everything else as is, click "Create Repository"
4. At this point an empty repository exists on GitHub. You can do one of two things:

If you have a local repository already created for the new code, do this:

1. Open git bash
2. Change to the local repository directory
3. `git remote add origin https://github.com/dopplerimager/NewRepoName`
4. `git push`

(Step 3 sets up the local repo to track the one on GitHub. Step 4 then uploads your local code to GitHub.)

If you don't have a local repository, do this:

1. Open git bash
2. Change to the directory where you want the new repo directory to be copied to
3. `git clone https://github.com/dopplerimager/NewRepoName`
4. `cd NewRepoName`
5. You can then start adding files to the repo using, e.g.:
6. `git add newcode.pro`
7. `git commit –am "Added awesome new code"`
8. `git push` (update the online repo).

Creating a new Local repository:
To create a new repository on the local machine, just create a new directory, then:

1. Open git bash
2. Change to the new directory (`cd path/to/dir`)
3. git init
4. You now have an empty repository. You add files to track using:
5. `git add dir/filename` (or `git add dir/*.extension`)
6. `git commit –am "Commit message"`

Step 6 is important: most git commands do not directly affect the current state of the repository, they just "stage" these changes, which means they are put in a queue to be carried out the next time you

type `git commit –am "Message"` (the `–a` means commit all staged changes, it is what I usually use).

Software Startup sequence:

sdi_main

The IDL entry point is `sdi_main` in `sdi_main.pro`.

Keyword arguments to `sdi_main`:
`settings` = (required) filename of an SDI settings file
`schedule` = (required if mode is auto) filename of a schedule file
`mode` = (optional – manual by default) 'manual' or 'auto'

There are usually scripts set up for each instrument to call this function through an IDL icon or similar, e.g.

AFA_Auto_SDI_Operation.pro

```
sdi_main, settings="C:\Users\sdi3000\setup\AFA_setup.sdi",
          schedule="C:\Users\sdi3000\setup\AFA_Schedule.txt", mode="auto"
```

AFA_Manual_SDI_Operation.pro

```
sdi_main, settings="C:\Users\sdi3000\setup\AFA_setup.sdi",
          schedule="C:\Users\sdi3000\setup\AFA_Schedule.txt", mode="manual"
```

The primary job of `sdi_main` is to create the IDL object '`XDIConsole`'. On creation, this object starts `xmanager`, which takes over control and waits for events. `sdi_main.pro` also containes the top-level event handlers `Handle_Event` and `Kill_Entry` (for object destruction). Calls to these functions are re-routed back to methods in the XDIConsole object (these methods are `Event_Handler` and `Kill_Handler` respectively).

XDIConsole

Starts up in the `Init` method, which takes the `settings`, `schedule` and `mode` arguments from `sdi_main`.

`Init` Sequence:

1. Locate plugins by searching through IDL search path for files of the form '`SDI*__define`'.
2. Create the SDI console GUI and create a menu containing the plugins which were found.
3. Create a top-level timer widget which is used to drive events.
4. Create `XDIWidgetReg` object, which is responsible for managing opened plugins (various utilities for finding them by name etc).
5. Load the settings file (this is implemented in the method `load_settings`).
6. Check that the settings file loaded correctly, if not, and we are running in manual mode, prompt for a new file, else die.
7. Load the console settings file – each plugin-type object stores things like geometry in a settings file. Not to be confused with 'settings file' which is required to passed as argument to `sdi_main`.
8. Compile the instrument-specific settings file.

9. Run the XXX_intialise method in the instrument-specific settings file, where the XXX is the instrument_name field in the main settings file, in the header structure.
10. Initialise the camera. This should eventually be ported into the instrument-specific file, but since all instruments do the same thing currently, this hasn't been done yet.
11. After initializing, we then start camera acquisition. Frames are grabbed in the timer_event method.
12. Read the position of the mirror motor using the instrument-specific file.
13. Set the filter position to be whatever was last stored in the settings file.
14. Register to receive timer events.
15. Compile found plugins (this seems to be necessary).
16. Start the top-level timer going.
17. Start xmanager.

Event Handling:

Events are intercepted first by Handle_Event in sdi_main.pro, but are immediately re-routed to Event_Handler in XDIConsole.

In Event_Handler (XDIConsole::Event_Handler), events are separated into timer events generated by the console gui and events which will be re-routed to plugins (including the console). Timer events are sent on to plugins which have registered to receive them (including the console).

The console handles timer events in XDIConsole::timer_event, and uses them to drive things like camera frame grabbing, crash testing, calculating solar elevation angle. Once frames are acquired they are passed onto plugins which have registered to receive frame events (plugins indicate the need for frame or timer events by setting inherited member variables self.need_timer = 1, self.need_frame = 1 in their init methods). These fields are inherited from a class called XDI_Base (XDIBase__define.pro) and do not appear explicitly in a plugins structure definition. All plugins need to inherit from XDI_Base.

Schedule Execution:

When the console decides it needs to execute a new schedule instruction, it calls XDIConsole::execute_schedule. Current schedule information is stored in console member variables self.runtime.schedule and self.misc.schedule_line. It first checks to see if the phasemap or steps/order need refreshing. It then calls the function schedule_reader (schedule_reader.pro), passing it the current schedule file line number (the line number of the last executed schedule command).

This function looks in the schedule file, beginning at the passed-in line number, and gives back the next schedule command and arguments based on the need to refresh phasemap steps/order, and the site latitude and longitude (to get solar elevation angle). It gets a reference to the console so it can retrieve the snr/scan. The rest of XDIConsole::execute_schedule executes actions depending on the returned command string. Currently implemented commands are:

- phasemapper
- stepsperorder
- spectrum

- `cameraset` (set camera exposure time and gain)
- `runscript` (`execute` an idl string)
- `mirror` (drive the mirror)
- `cal_switch` (select calibration source)
- `filter` (select filter)
- `wait` (execute IDL `wait` function)
- `log` (write a string to the console log)

In general, **the schedule file should not be changed while the SDI is running from it**. Problems arise due to mismatched line numbers etc, so if you want to update the schedule file, you should first change the SDI mode back to 'manual', edit and save the schedule file, then switch the mode back to 'auto'. The console will then start reading from the start of the schedule file again, and it should all work OK.

When the console determines that it needs to refresh the phase map or steps/order value based on these fields in the settings file:

```
etalon.phasemap_refresh_hours
etalon.nm_per_step_refresh_hours
```

it asks the `schedule_reader` to look in the schedule file to see if any commands of the form:

```
%% stepsperorder: [632.8, 660, 730, 30, 4, 50, 0.18]
&& phasemapper: [0, 1, 632.8, 632.8, 50, 0.18, 3]
```

are present. These commands are the corresponding refresh commands that will be run when the console determines its phase map or steps/order are out-of-date. Unfortunately you have to look into the plugins themselves to see what these arguments represent (look in the `auto_start` method), or at the top of many schedule files the arguments are listed. A template schedule file (with these arguments spelled out) is in the `idl/` sub-directory.

Schedule Script Syntax

The schedule file can contain the following control directives:

`ifsea: [ low, high ] [ loop | cont ]`
This command sets up a loop based on the current solar elevation angle (sea). When encountering the command `ifsea: [ low, high ] [loop]`, if the current solar elevation angle lies between the given limits, execution continues on the following line (it enters the body of the loop). Upon encountering a `ifsea: [ low, high ] [cont]` command, if the elevation angle is between the given limits, control goes back to the start of the loop (it looks for the previous `ifsea` command, so these cannot nest). If the elevation angle is outside the given limits, control resumes on the line following the `ifsea: [ low, high ] [cont]` command.

`ifsnr: [ low, high ] [ begin | end ]`
This directive tells the schedule file to only execute the code between the `ifsnr: [low,high] [begin]` … `ifsnr: [low,high] [end]` pair if the signal-to-noise ratio per scan at 557.7nm lies between the given limits.

`ifut: [ low, high ] [ begin | end ]`
This directive tells the schedule file to only execute the code between the `ifut: [low,high] [begin]` … `ifut: [low,high] [end]` pair if the current universal time lies between the given limits.

Note that the last two directives do not set-up loops – after seeing an `ifut/ifsnr:[low,high] [end]` execution continues on the following line.

Adding new scheduled commands:

If you need to implement new schedule file commands, you want to add code to the `XDIConsole::execute_schedule` method. As an example, here is the filter command, used to select a new filter:

```
if command eq 'filter' then begin
      filter_number = fix(args(0))
      current_filter = self.misc.current_filter
      log_path = self.logging.log_directory
      call_procedure, self.header.instrument_name + '_filter', $
                      filter_number, log_path = log_path, self.misc, self
      self.misc.current_filter = filter_number
      self -> save_current_settings
      self -> log, 'Selected Filter ' + string(filter_number, $
                  f='(i0)'), 'Console', /display
endif
```

The command will be a string with spaces removed, and **remember that each argument returned by the schedule reader is a string**, so in the example above, the `args[]` array is a string array, and since the filter number is an integer, it needs the `fix(args[0])` to get a number.

Plugin Startup:

Handled by `XDIConsole::start_plugin`. Plugins can be started either by clicking in the drop-down menu from the console gui, or through a schedule file command, in which case the plugin is being 'auto-started' (each plugin needs an `auto_start` method to handle this). The first part of `start_plugin` determines which of these two scenarios apply. It then has a special case for if the plugin is a spectrum plugin and is not being auto-started, in which case it asks for a wavelength, since this is required in order to actually create this particular plugin.

It then does the following:

1. Build a structure containing some info about the current execution environment, and some relevant variables, which gets passed to every plugin for initialization.
2. Check if saved settings exist for the plugin, if they do then restore them, the restored data is in a structure called `restore_struc`.
3. Increase the object count (object count is used to provide unique id's for plugins).
4. Create the new object instance.
5. Check to see what timers the created plugin needs, and register with the manager object.
6. Clear any frames accumulated during this time.

Plugins:

A plugin template is located in the `idl/` sub-directory (`Template_Plugin.txt`). Copy this and rename to `SDIMypluginname__define.pro` in order to use.

Plugins are IDL objects. In order for them to work with the SDI, they must inherit from `XDIBase`, which defines things like geometry, some handles to the console and widget manager objects, etc. Plugins also need to have a variable called `id` in their structure definitions – this is used to hold the widget id of the plugin's main window (I am not sure why this was never put into `XDIBase`).

Inside the `Init` method of a plugin, the plugin can tell the console that it wants to receive timer events, or frame (new camera image) events, or both. It does by first setting the following flags:

```
self.need_frame = 1
self.need_timer = 1
```

And by returning these fields in its `::get_settings` method (see below). When either of these fields are set to 1, the plugin also needs to define the corresponding methods. These methods are:

```
pro PluginName::frame_event, image, $ ;\\ latest camera image
                             channel  ;\\ current scan channel
end

pro PluginName::timer_event ;\\ no arguments
end
```

The console will call these events, depending on which of the two flags have been set, when a timer event is generated or when a new camera image is received.

The plugin also needs to define a method to fill up a structure with settings that it wants to save, in order to restore them when it is next instantiated. The method looks like this:

```
function SDIVidshow::get_settings

  struc = {id:self.id, $
           need_frame:self.need_frame, $
           need_timer:self.need_timer, $
           geometry:self.geometry, $
           scale:self.scale, $
           scale_fac: self.scale_fac, $
           exp_time:self.exp_time, $
           crosshairs:self.crosshairs,
      crosshairs_point:self.crosshairs_point, $
           grid:self.grid}

  return, struc
end
```

This example is from the Vidshow plugin, each of the fields in the struc are fields from the plugins own class structure which it wants to restore the next time it is started up. **The fields in red should be present for every plugin**. A plugin uses these restored settings inside its `Init` method. The structure

from `get_settings` is passed back to the plugin's `init` method as a keyword `restore_struc`, and a flag is set inside the `data` structure, which is also passed to `init` via a keyword. For example, from Vidshow:

```
function SDIVidshow::init, restore_struc=restore_struc, $   ;\\ Restored settings
                           data=data                        ;\\ Misc data

       self.need_timer = 0
       self.need_frame = 1
       self.manager  = data.manager
       self.console  = data.console
       self.palette  = data.palette
       self.obj_num  = string(data.count, format = '(i0)')
       self.xdim     = data.xdim
       self.ydim     = data.ydim

       if data.recover eq 1 then begin
              ;\\ Saved settings
              xsize  = data.xdim
              ysize  = data.ydim
              xoffset = restore_struc.geometry.xoffset
              yoffset = restore_struc.geometry.yoffset
              self.scale = restore_struc.scale
              self.scale_fac = restore_struc.scale_fac
              self.grid = restore_struc.grid
              self.crosshairs = restore_struc.crosshairs
              self.crosshairs_point = restore_struc.crosshairs_point
       endif else begin
              ;\\ Default settings
              xsize = data.xdim
              ysize = data.ydim
              xoffset = 100
              yoffset = 100
              self.scale_fac = 0.005
       endelse

       ;\\ other stuff here….
end
```

The restoration code above is shown in blue.

Adding fields to the settings file:

The settings file contains a set of structures (`etalon, camera, header, logging, misc`) plus a structure defining the com ports for different pieces of hardware. When adding or removing fields from these structures, note that the definitions actually occur in two places and need to be identical: they occur in `XDIConsole__define.pro` (down the bottom, in the `XDIConsole__define` method) and `edit_console_settings.pro`. If you update one, don't forget to update the other or problems will ensue. Each of the above structures contains a field called `editable`, which is a vector containing the indices of all fields which are meant to be edited by the user. Be sure to update this field if you add to one of the structures and you want that field to be editable. It is easier to add fields at the end of the structure (but before the `editable` field) so that you don't need to adjust all the indices. Also note that any default values placed into the definitions inside `XDIConsole__define.pro` will **not be preserved** – these definitions occur inside a class definition, and don't mean anything, all types will get their IDL-default initializers (at least this is how IDL 6.2 worked).

Adding a new field should be easy as adding it into one of the existing structures in both `edit_console_settings.pro` and `xdiconsole__define.pro` (I just tried it and it worked OK, however you may need to do a `.reset_session` in order to clear any previous definition of the structure).

If, for example, you add a new field to `misc` (in `edit_console_settings`), then compile and run `edit_console_settings`, assuming you made the field editable, it should appear in the tree of fields. If you then load a settings file which does not define that field, it will still load, but that field will not be updated with new information (since none was defined in the file).

Instrument Specific Files:

The settings file contains a field in the `header` structure called `instrument_name`, which should be a string name without spaces. **The name in this field is important because it determines where calls to certain instrument/hardware specific functions are directed**. I'll use the Poker Flat instrument as an example. In the settings file for this instrument, `header.instrument_name = "PokerFlat"`. There is a corresponding file called `PokerFlat_initialise.pro`, which contains a set of routines whose names begin with `PokerFlat_`. Each instrument has a file like this, and the following procedures need to be defined within it (again using Poker Flat as an example):

```
;\\ Called on shutdown, close ports etc. here.
PokerFlat_cleanup, misc, $
                   console

;\\ Called when we eant to change the calibration source.
PokerFlat_switch,  source, $
                   misc, $
                   console, $
                   home=home

;\\ Called when we want to change the filter.
PokerFlat_filter,  filter_number, $
                   misc, $
                   console, $
                   log_path=log_path

;\\ Called to update etalon plate separation.
PokerFlat_etalon, dll, $
                  leg1_voltage, $
                  leg2_voltage, $
                  leg3_voltage, $
                  misc, $
                  console

;\\ Called every time an image is acquired form the camera.
;\\ Do background subtraction etc. here.
PokerFlat_imageprocess, image

;\\ Called on startup, open com ports and stuff here.
PokerFlat_initialise, misc, $
                      console
```

Note that `PokerFlat_initialise` is declared last for compilation reasons, so stick with this layout. Also, the argument lists are not very consistent between the different procedures, and some are

redundant, but it has evolved this way ☺. In each case, the `misc` argument contains the `misc` data structure from the settings file, while the `console` argument is the object reference of the console, allowing the procedures to call any of the methods defined in the console. Other arguments are fairly self-explanatory. There is a template instrument-specific file in the `idl/` sub-directory called `default_initialise.pro`.

Building the DLL Wrapper (SDI_External.dll)

To build the DLL wrapper, I have used the MinGW compiler (GCC for Windows) and the Code::Blocks IDE. This is probably the easiest way to do it. You can download Code::Blocks bundled with the MinGW compiler, from http://www.codeblocks.org/downloads/ (http://www.codeblocks.org/downloads/26 at the moment), making sure to select the codeblocks-10.05mingw-setup.exe version or similar.

Now get the code (open git bash, change to the directory where you want the sdiexternal repo to go):

```
git clone https://github.com/dopplerimager/sdiexternal
```

The repository in `sdiexternal/` will contain a directory called `build/`. In here, there will be a codeblocks project file called `SDI_External.cbp`. Open this up in the Code::Blocks IDE, select the build target (Build -> Select Target -> Release), then Build -> Rebuild. The resulting dll file will be in `sdiexternal/build/bin/release`. Replace the old SDI_External.dll (wherever it is, usually `sdi/bin`) with the new one. You probably also want to update the copy of SDI_External.dll in the GitHub `sdi` repository. To do this, just copy the dll to a local `sdi` repo, (put it into `sdi/bin`), do a git commit (`git commit –am "Updated DLL"`), then `git push`.

Real-time Analysis:

The real-time analysis software runs on the SDI_GI_SERVER machine (which also acts as a data store). The IDL routines are all located at:
`C:\RSI\IDLSource\NewAlaskaCode\Routines\SDI\Monitor`
The software can be started by typing `sdi_monitor` at the IDL command prompt. The basic operation is fairly simply, the `sdi_monitor` routine simply enters an infinite loop, and regularly checks for new data files (snapshots) in the `C:\FTP` directory. These snapshots contain the last acquired spectra (the last completed exposure) from each instrument, along with a minimum of metadata. The monitor routine fits these spectra, and generates plots of the spectra overlaid on sky maps of the temperature or intensity or signal/noise. The routine also stores the latest fits in time-series data files, so it can plot time-series of winds and temperatures from each site. These time-series are stored in the /Timeseries sub-directory. Only a fixed number of time-series data are stored – currently 1000 exposures from each instrument. Older exposures simply fall off the end of the array, and are forgotten.

The monitor routine also FTP's some images to the fulcrum server as they are generated.

Many of the display options are located in separate IDL files – `sdi_monitor_snapshots.pro` and `sdi_monitor_timeseries.pro`. Look in here for many of the hard-coded color table and scaling options. You can edit and recompile these files while the monitor routine is running, and the new changes will be picked-up on the next refresh.