**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).

**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). This function 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
* camaraset (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)

If you need to implement new schedule file commands, this is where you do it. Here is the example for the filter command:

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

**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.

**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. 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).