Tutorial: New Devices in Stage 2.0.0a

Michael Janssen

September 6, 2005

Abstract

This document describes the process of adding a new driver to Stage version 2.0.0a. The reader is provided step-by-step instructions including files to edit, functions which must be created and their purposes, and testing which should be done.

1 Introduction

A device in stage has a number of properties, which can be read and set via the internal *stg_model_set_property(stg_model_t*, char*, void*, int)*, *stg_model_get_property(stg_model_t*, char*, void*)*, and *stg_model_get_property_fixed(stg_model_t*, char*, void*, int)*. Properties are referred to by strings. The fixed function will return NULL if the property does not contain the size given. This is useful for storing whole structures as a property in the model and retrieving them later without worrying about overflow errors.

2 Creating a Driver

There are X steps to creating a stage driver:

We will use examples from the compass driver which is included in the UMN Player/Stage distribution.

2.1 Adding the data and configuration data to Stage

The first decision which must be made is how the data from the device is going to be stored within stage. This must then be added via a struct to the `stage.h` file. All data which should be communicated to player clients should included in this model. In our example, the only data which needs to be passed to a player client is the direction to "north".

The configuration of the device is also stored in the `stage.h` file. The data which should be stored over time, generally used to describe the configuration of the sensor. In our example, the only configuration data which needs to be stored is the direction which is considered to be north.

Naming conventions in the `stage.h` file are **stg_XXX_config_t** for the configuration and **stg_XXX_t** for the data.

```
...
// COMPASS MODEL --------------------------------------------------------
/** @defgroup stg_model_compass Compass
    Implements the compass model
    @{ */
/** compass config packet
 */
typedef struct
{
  double north;
} stg_compass_config_t;

typedef struct
{
  double to_north;
} stg_compass_t;

/**@}*/
..
```

2.2 Creating the Model File

The first thing which any Stage device should have is a model. This is the internal representation which is used by Stage in order to actually simulate. There are no restrictions on how complicated or simple a model can be, however a number of functions must be created for use by the Stage framework:

- *XXX_init(stg_model_t*)*
- *XXX_startup(stg_model_t*)*
- *XXX_shutdown(stg_model_t*)*
- *XXX_update(stg_model_t*)*
- *XXX_load(stg_model_t*)*

In addition to these functions, generally included in the model file are the functions to draw on the Stage GUI the data which is being returned and the configuration of the sensor:

- *XXX_render_data(stg_model_t*, char*, void*, size_t, void*)*
- *XXX_unrender_data(stg_model_t*, char*, void*, size_t, void*)*
- *XXX_render_cfg(stg_model_t*, char*, void*, size_t, void*)*

These functions are setup to be called in *XXX_init* function.

In these functions, the name of the device generally replaces the XXX in the function names above. None of these functions are required to be named in this way, but they are required to exist and in current practice they are all named this way.

2.2.1 XXX_init

This is the initialization function, which is called by the generic model to initialize the specific model. It is called when the model is created from the worldfile. This function sets up all of the rest of the callbacks which are used in the model. It also should setup default properties for the model if they are not overridden in the worldfile.

```
int compass_init ( stg_model_t* mod ) {
  // override the default methods
  mod->f_startup = compass_startup;
  mod->f_shutdown = compass_shutdown;
  mod->f_update = NULL; // installed at startup/shutdown
  mod->f_load = compass_load;

  // remove the polygon: sensor has no body:
  stg_model_set_property( mod, "polygons", NULL, 0);

  stg_geom_t geom;
  memset( &geom, 0, sizeof(geom));
  stg_model_set_property( mod, "geom", &geom, sizeof(geom));

  stg_color_t color = stg_lookup_color( "magenta" );
  stg_model_set_property( mod, "color", &color, sizeof(color) );

  stg_compass_config_t cfg;
  memset(&cfg, 0, sizeof(cfg));

  cfg.north = STG_DEFAULT_COMPASS_MAGNORTH;

  stg_model_set_property( mod, "compass_cfg", &cfg, sizeof(cfg));

  // start with blank data
  stg_compass_t data;
  memset(&data, 0, sizeof(data));

  stg_model_set_property( mod, "compass_data", &data, sizeof(data) );

  // Menu is moved to here?
  stg_model_add_property_toggles( mod, "compass_data",
      compass_render_data,
      NULL,
      compass_unrender_data,
      NULL,
      "rangecom data",
      TRUE );

  return 0;
}
```

2.2.2 XXX_startup

This startup function is called by the generic model when the first subscriber subscribes. It generally sets up the updating function and any other properties of the sensor which would change when it is powered on.

```
int compass_startup( stg_model_t* mod )
{
  PRINT_DEBUG( "compass startup" );

  mod->f_update = compass_update;
  //mod->watts = STG_RANGECOM_WATTS;

  return 0;
}
```

2.2.3 XXX_shutdown

This shutdown function is called by the generic model when the last subscriber unsubscribes. It usually removes the update function from the model so that unneeded processing is not done when noone is using the device, and changes other properties which would change when the simulated sensor powers off. It also generally unsets the data property so the stage GUI will not continue drawing the sensor data.

```
int compass_shutdown( stg_model_t* mod )
{
  mod->f_update = NULL;
  //mod->watts = 0.0;

  // this will unrender the data
  stg_model_set_property( mod, "compass_data", NULL, 0 );

  return 0;
}
```

2.2.4 XXX_update

This function updates the sensor to represent a change in the world. It is called whenever the world is changing. The default update function simply updates the position of a model based on it's velocity properties, allowing for moving objects in the world without them being a robot.

This function is generally the most differing between models, because most of the actual simulation happens here.

```
///////////////////////////////////////////////////////////////////////////
// Update the data
int compass_update( stg_model_t* mod )
{
  PRINT_DEBUG( "compass update" );

  if( mod->subs < 1 )
    return 0;

  stg_compass_config_t* cfg =
    stg_model_get_property_fixed( mod, "compass_cfg", sizeof(stg_compass_config_t) );
  assert(cfg);

  stg_pose_t mypose;

  stg_model_get_global_pose( mod, &mypose );

  stg_compass_t data;
  memset( &data, 0, sizeof(data));


  data.to_north = NORMALIZE(cfg->north - mypose.a);
  stg_model_set_property( mod, "compass_data",
      &data, sizeof(stg_compass_t) );

  return 0;
}
```

2.2.5 XXX_load

Called when the model is created, after the *XXX_init* function, this function reads properties relating to the configuration of the model from the world file. This function is special in that if it exists in a model it is called in conjunction with the default loading operations, which include the position of the model and the outline and/or polygons which represent the model. It can override any of those functions however because it is called after those properties are read by the default handler.

```
void compass_load( stg_model_t* mod )
{
  stg_compass_config_t* now =
    stg_model_get_property_fixed( mod, "compass_cfg", sizeof(stg_compass_config_t));
  assert(now);

  stg_compass_config_t cfg;
  memset( &cfg, 0, sizeof(cfg) );

  cfg.north = wf_read_angle(mod->id, "compass", now->north );

  stg_model_set_property(mod, "compass_cfg", &cfg, sizeof(cfg));
}
```

2.2.6 XXX_render_data

This function draws the data returned on the GUI. In order to draw on the GUI, you need a **stg_rtk_fig_t** pointer, which can be created through the *stg_model_fig_create* function, which also places a copy in your model at a property name you specify, normally "XXX_data_fig". There are a number of drawing commands available in the stk which can all be used. The fig created for your model has it's origin at your model and rotated appropriately.

This function is passed the data of the object through the data argument, so it does not need to be retrieved separately.

```
int compass_render_data( stg_model_t* mod, char* name, void* data, size_t len, void* use
{
  stg_rtk_fig_t* fig = stg_model_get_fig( mod, "compass_data_fig" );

  if (!fig)
    fig = stg_model_fig_create( mod, "compass_data_fig", "top", STG_LAYER_NEIGHBORDATA )

  stg_rtk_fig_clear( fig );
  stg_compass_t *direction = (stg_compass_t*) data;

  stg_rtk_fig_arrow( fig, 0, 0, direction->to_north, 1.0, 0.10 );
  return 0;
}
```

2.2.7 XXX_unrender_data

This function clears the data which is drawn on the canvas, usually because the device is not subscribed to anymore.

```
int compass_unrender_data ( stg_model_t* mod, char* name, void* data, size_t len, void* 
  stg_model_fig_clear( mod, "compass_data_fig" );
  return 1; // cancel callback
}
```

2.2.8 XXX_render_cfg

This function should draw on the canvas any information which can be displayed graphically related to the configuration of the sensor. This would include for example the range of the sensor or the view frustrum. The example compass driver has nothing it could display which could be considered configuration so it does not include a render configuration function.

Like the *XXX_render_data* function, this function needs a figure to draw on, which can be created using the *stg_model_fig_create* function, which also places a copy in your model at a property name you specify, normally "XXX_cfg_fig".

2.3 Creating the Player Interface

The `p_XXX.cc` file is where the heart of the conversion from stage internal simluation to the player drivers. It is required to consist of specific functions, and be extended from a common object **InterfaceModel**. The important thing to remember here is to switch the byte-ordering of the data which is being placed into the player data structures by calling *ntohl*.

The compass example should be similar to the rest of the drivers (it simulates a position driver, following the example of a p2os driver). The Publish function is called when player queries stage for new data. It should call PutData at the end. The Configure function is called when the Player driver gets a configuration request - the configuration message is passed verbatim to the function along with the length.

```
#include "p_driver.h"

extern "C" {
int compass_init( stg_model_t* mod );
}

InterfaceCompass::InterfaceCompass( player_device_id_t id,
        StgDriver* driver,
        ConfigFile* cf,
        int section )
  : InterfaceModel( id, driver, cf, section, compass_init )
{
  this->data_len = sizeof(player_position_data_t);
  this->cmd_len = 0;
}


void InterfaceCompass::Publish( void )
{
  player_position_data_t pdata;
  memset(&pdata, 0, sizeof(pdata));


  stg_compass_t *sdata = (stg_compass_t*)
    stg_model_get_property_fixed( this->mod, "compass_data", sizeof(stg_compass_t));

  pdata.yaw = ntohl((int32_t)(RTOD(sdata->to_north)));

  this->driver->PutData( this->id, &pdata, sizeof(pdata), NULL);

}

void InterfaceCompass::Configure( void* client, void* buffer, size_t len )
{
  printf("got position (compass) config request, ignoring\n");

  if (this->driver->PutReply( this->id, client, PLAYER_MSGTYPE_RESP_NACK, NULL, 0, NULL)
    DRIVER_ERROR("PutReply() failed");
}
```

3 Connecting it to Stage

Now we need to connect the new driver to the rest of Stage. This is done by adding a few lines to the `p_driver.cc` file in this switch:

```
      switch( player_addr.interf )
      {
         ...
      }
```

If the player device that you are creating is not simluated by anything in Stage yet, you can find the code constant which is used to describe your device in the player code. Usually they are named **PLAYER_XXX_CODE** where XXX is the Player driver which is being simulated. You can simply add the case that you are using to the switch in the format of the other files.

If your device is already simulated by some other model in stage, you will have to include some way to distinguish it from the "default" model which is created for that device. The current way to do this is by adding a key in the config file to distinguish it from the default model. Then you add a check for the key before the default. This is what we needed to do with the compass model because while it simulates the position interface, there was already a position model in stage:

```
...
  case PLAYER_POSITION2D_CODE:
    if (cf->ReadDeviceId( &player_addr, section, "provides", 0, d, "compass") == 0) {
      printf (" (compass position) ");
      ifsrc = new InterfaceCompass( player_addr, this, cf, section );
    } else {
      printf (" (normal position) ");
      ifsrc = new InterfacePosition( player_addr, this,  cf, section );
    }
    break;
...
```

You must also add your InterfaceXXX to the `p_driver.h` file, so that it can be called from the switch:

```
...
class InterfaceCompass : public InterfaceModel
{
 public:
  InterfaceCompass( player_device_id_t id, StgDriver* driver, ConfigFile* cf, int sectio
  virtual ~InterfaceCompass( void ){ /* TODO: clean up*/ };
  //virtual void Command( void* buffer, size_t len );
  virtual void Configure( void* client, void* buffer, size_t len );
  virtual void Publish( void );
};
...
```

This is usually added at the end of the file.

4 Getting the driver compiled

Stage uses the automake system, so in order for your driver to be compiled into stage, you need to edit the `Makefile.am` file. Add your model file (in our example model_compass.c) to the list of files in the libstage_la_SOURCES variable, and your interface file (in our example p_compass.cc) to the libstage-plugin_la_SOURCES variable. Don't forget the continuation marks on the lines, or you will most likely be baffled at errors provided when you try to compile.