HSpace Tutorial

Edited by Mark Hassman

Revised: March 29, 2004

Table of Contents

HSpace Preview

What is HSpace?

HSpace is an add-on for a MUSH server that provides a 3 dimensional space simulation. HSpace provides full combat and navigation support for space vessels and the ability to develop a realistic space environment with planets, wormholes, black holes, nebula, and asteroid belts. A full ship vs. ship combat system is included with the flexibility to create ships with many different characteristics and weapons. A softcode interface to the system provides the capability for administrators to build custom tools for the design and creation of ships, navigational aids, economy systems, or just about anything imagineable.

HSpace comes from the name Hemlock Space, and it was first developed on the Hemlock MUSH series of games. It was released in 1997 and has been increasing in popularity ever since. Now, it is written in a more object-oriented gaming language -- C++. This allows even more realistic and flexible space modeling to enhance your gaming experience!

HSpace is primarily developed for the PennMUSH and TinyMUSH 3 servers. However, with some programming experience, it can be ported to work with other MU* servers.

Where can I get it?

HSpace is distributed from www.hspace.org and is also available from an anonymous ftp mirror at moosh.net. A forum system is available for user of the systems at forums.hspace.org. The forums contain a variety of topics and a place to ask for help. The forum system is also the location where new releases are announced. Additional websites also provide tips, code suggestions, and additional information.

How do I get it installed?

A file is included with the distribution that details installing the system. First you will want to unpack the distribution to get to the installation instructions. To unpack the distribution, you'll need to have an application capable of handling tar and gzip. These are included with most UNIX style systems or can be found as shareware for Windows machines, such as WinZip. Once the package is expanded, look for the file INSTALLATION_GUIDE.txt. This file contains the latest installation instructions in the next section.

If HSpace is not ported directly to your MU* server of choice, you will need to port the system to work with the server. Such ports are beyond the scope of this document but you will need to understand both the code for the game server and HSpace itself. Porting will require knowledge of C/C++ for most game servers and the ability to read and understand the core software. If you do not have those skills, it is highly suggested you find someone who does before undertaking the task.

What's next?

Once HSpace is installed and operating, you can begin the process of filling out the space environments for your game and adding ships. The next few sections of this document aim to guide you through that process.

Installation Guide

What is HSpace?

If you are familiar with MUSH, then you probably have some idea of what HSpace is. If you're not familiar with MUSH and online text games (that's what a MUSH is), then you probably have no use for this package.

HSpace (Hemlock Space, from the Hemlock series of games) is a sort of "plug-in" to online text games, adding to the game a virtual space environment with ships that can navigate and engage in combat and a variety of other space-like things. Since most text-based game engines (e.g. PennMUSH or TinyMUSH) only provide a basis for creating an online text game, to create a space game you either need to develop your own space engine, choose not to have any space flight at all, or download and install a package like HSpace. You have obviously chosen the latter of the three. If you are still interested in using HSpace, then continue reading the sections to follow.

Supported Game Engines

This distribution of HSpace (there are other mutants available) currently only works with PennMUSH and TinyMUSH 3. HSpace was built in such a way that it can be easily adapted to other MUSH engines similar to PennMUSH.

Package Contents

You have unpacked the HSpace package, which contains a single "hspace" directory with the HSpace source code and this text file that you are reading. There is a Makefile in the same directory, which you will use to build the HSpace library.

Building HSpace

First, you will need to unpack and compile your game engine, be it PennMUSH or whatever. It is important for you to compile your game without HSpace to be certain that you can even get that far. Build your game, start it up, and get it running well. If you install HSpace without first testing your game without it, you won't be able to distinguish normal game problems from problems that could be caused by the space system!

Ok, you know your game works, right? Shut down your game, and let's build HSpace.

To build HSpace, you need to do the following things:

For PennMUSH:

1. Move your hspace directory into your pennmush/src directory, so that your hspace directory should be at pennmush/src/hspace.

2. Edit your pennmush/config.h and find the line that reads "#define HASATTRIBUTE /**/" Put a /* in front of that line if it's not already there.

3. Go to the hspace directory, and type make.

If all goes well, it should compile. At the beginning, you may see some warnings that certain dependency files do not exist, and that is quite all right. You should see several lines like "Generating dependencies for ..." and "Compiling ..." If you see any errors, you have compilation problems that were not foreseen when this package was created. If that happens, go to the sectioned titled "WHERE TO GET HELP" in this guide.

When your package is done building, you will see a file called in the hspace directory named "libhspace.a." This is the fully compiled HSpace code that will need to be "plugged in" to your game engine. Go to the section titled "INSTALLING HSPACE FOR <X>," where <X> is your game type.

Installation for PennMUSH

You will need to modify two files for PennMUSH to add HSpace: src/local.c and src/Makefile.

Make the following change to src/Makefile (in the src directory, not the top level directory!):

Change: CLIBS=$libs $cryptlib
To: CLIBS=-lpthread -Lhspace -lhspace -Lhspace/hsnetwork -lhsnetwork -lstdc++ $libs $cryptlib

Note that this assumes you have your hspace package directory in the PennMUSH src directory! If this is not the case, then you should have -L<path to libhspace.a> (e.g. -L../spacestuff/hspace).

Make the following changes to src/local.c, but do not include the <--- and the text after it!:

```
local_startup()  <--- Find this
{
        hsInit(); <--- Add this
}

local_dump_database() <--- Find this
{
        hsDumpDatabases(); <--- Add this
}

local_shutdown() <--- Find this
{
        hsShutdown(); <--- Add this
}

local_timer() <--- Find this
{
        hsCycle(); <--- Add this
}
```

When you have made these changes, change to your top-level PennMUSH directory, and type 'make install', which should compile local.c again and link the libhspace.a file with your PennMUSH engine.

In the PennMUSH game directory, make a directory named "space," and copy the hspace/hspace.cnf file to this directory. Edit the hspace.cnf, and change things as you please.

Copy all files in the hspace/help_files directory to your pennmush/game/txt/hlp directory. From pennmush/game/txt type 'make' to rebuild the help files.

Restart your game, and you should have HSpace commands available. Type 'help hspace' for the HSpace help files.

Installation for TinyMUSH3

While the code for HSpace includes support for the TinyMUSH 3 server, the integration steps for inserting the system into the TM server is much more complicated. To avoid the necessity of complicated patches and updates, a version of the TM server plus the addition of the HSpace source is distributed as a full package. The current release support TM3.1p11 and HSpace 4.3.4. It is downloadable from http://moosh.net/pub/mush/hspace/TinyMUSH-Ports/ and can be built utilizing the standard TM3 build process. All necessary changes have been applied to the makefiles, configure scripts and game directories. The distributed netmush.conf file contains configuration directives that must be included in the game configuration file.

The first step is to download the tar, gzipped package such as tm-hspace-1.0.13.tgz. Unpack this archive by:

```
> tar –czvf  tm-hspace-1.0.13.tgz
```

This produces a directory called tinymush. Move into that directory and build the entire system by running the build script:

```
> ./Build
```

This will produce the necessary executables and prepare the game for running. After the compilation is finished, you modify the configuration files and game setup just as you would the normal TM server.

If you need to apply updates, the hspace source is located in tinymush/src/hspace. You can install the latest release of HSpace if you so desire by unpacking the current release into that directory. You will then need to modify the HSpace Makefile and change the MUSHTYPE definition from PENNMUSH to TM3. You can then utilize the Build script once again to produce a new executable.

Where to Get Help

HSpace has been in use for quite some time and has had time to evolve, but the package is free, so to some extent you must expect there could be some problems that take time to sort out. One of the main problems could be that the package doesn't compile for you. This would not be a surprise, and the developers will need to know why it failed. Another problem could be that these instructions were not clear enough, so the developers will need to know that as well. Or maybe you just have some burning questions on your mind and want to ask them. In any case, here's the address where you can contact the developers of this package, and this package only!

gepht@hspace.org

Please include the package version, found in hsversion.h, the operating system you are using for your game server, and a decent description of the problem or question.

Additional information can be garnered from http://www.hspace.org!

HSpace Windows Installation Guide

Introduction

This document describes briefly how to build and use HSpace on a computer running the Windows operating system. This discussion only accounts for the PennMUSH game engine, since HSpace officially only works with that engine.

Where to start?

First, you should begin by downloading, building, and firing up the game server on your machine. This probably means downloading PennMUSH and compiling it with Microsoft Visual Studio or some similar compiler that does the job. Get everything working before you go adding HSpace, which is our advice on both Windows and UNIX.

Adding HSpace to Your Game

You have probably already downloaded and unpacked HSpace from www.hspace.org, which is why you are reading this file. Be sure the hspace directory from the HSpace package is located in the pennmush/src directory, so that the path to HSpace is pennmush/src/hspace. In the hspace directory, you will find the hspace.dsp project file for Microsoft Visual Studio. If you are using some other compiler (like gcc and cygwin), you may find the Makefile useful in that same directory. Add the dsp to your PennMUSH workspace (or compile the Makefile to produce hspace.lib). If you are using Visual Studio, choose the 'Project' menu, and then 'Dependencies'. Make the hspace project a dependency of the pennmush project. Thus, when you compile pennmush, it will first compile hspace, then pennmush, and then link the two together at the end.

!! Read the HSpace installation guide that comes with HSpace and follow the instructions in that guide for modifying PennMUSH so that it properly talks to HSpace.

And now?

Provided you have followed the instructions above and received no compiler errors, you should be good to go. HSpace is developed natively on Windows and ported to UNIX, so there should be no compatibility issues with Windows. When you start up your game, HSpace should work as it is supposed to. Try logging into your game and typing @space. You should see a full HSpace statistics display. If not, then something has gone wrong, and you should contact **gepht@hspace.org**.

HSpace Celestial Objects

Everything in HSpace is represented as an HSpace object. There are a variety of types that these objects can take. As of version 4.2.2, the following objects are supported: Ships, Missiles, Planets, Wormholes, Black Holes, Nebula, and Asteroid Belts. See 'HS-TYPES' for the current list of types and type identifiers. Each type of object contains a core set of attributes common to all objects and may contain additional attributes that specify details specific to only that type. The core set of object attributes are the name of the object as viewed from space, its size from 1 to n, the X, Y, and Z coordinates of the object, and the universe identification number of the universe it belongs too.

Additionally, HSpace provides support for multiple universes and territories within those universes. All objects must belong to a universe. It is common to have at least two universes: One for the actual space environment and another for simulators where player's can practice their piloting skills without being killed in combat.

Universes

Since every HSpace object must belong to a Universe, it is the logical starting point for construction the space environment. A single room represents every universe. Thus, a room must exist for every universe. This room is used for when ships are actively flying through space. Administrators can also look at the contents of the room to determine what objects are active in that particular universe. Games can have one or as many universes as they deem appropriate.

To create a universe, first dig the room that will represent it.

@dig Space Room

Note the DBREF of the room after the command completes. The DBREF is needed when adding the universe into the HSpace system. To add the universe to HSpace, do the following:

@space/newuniverse <room dbref>

You can use *@space* or *@space/list universes* to verify the new insertion. Beware, you must not delete the room representing the HSpace universe. Doing so will cause the HSpace system to fail.

Universes can also be removed from the HSpace system. To be removed, the Universe must no longer contain any HSpace objects. Follow the instructions for deleting other object types prior to deleting the universe. Once all objects are removed, do the following:

@space/deluniverse <universe id>

The <universe id> can be determined by using *@space/list universes*. After the universe is removed from HSpace, the room representing the universe can also be purged with *@destroy*.

Celestial Bodies: Planets, Wormholes, Black Holes and Asteroids

Having a universe is all well and good and quite necessary, but without something in the universe, its just a big blob of three dimensional space. The next step is to insert the HSpace objects that exist within our new universe. These objects are the suns, planets, moons, etc. that make up the space environment.

HSpace does not contain representations for galaxies or solar systems. Those can be accomplished by the X,Y,Z coordinates of the suns, moons and planets. Depending on your needs, you may have only a few celestial objects in your universe. This could be a simple solar system or a dozen galaxies with a dozen solar systems. Similarly, HSpace does not differentiate between planets and suns. Suns are essentially just large planets.

Considering you've read this far, I'll assume you have some idea of what space will contain. Throughout the rest of this example, I'll use the Earth's solar system as its easy to visualize for us mere humans. Many games, however, will not use Earth based systems so this is merely an example.

Earth is in the Milky Way galaxy. It is in a solar system with 9 planets, several with one or more moons orbiting around the Sun. A black hole is at the center of the galaxy. An asteroid belt lies between Mars and Jupiter. Throughout the galaxy are many more stars: some with planets, some without. One or more Nebulae may be spread throughout the galaxy.

Planets

Yeah, yeah, you know all that already so let's place them into our new universe. First off, every object to be inserted into HSpace must also be represented by an actual game object. Thus, it is necessary to first create the game object that will represent the HSpace object in the MUSH server. These objects are THINGS in MUSH terminology and created via *@create*. HSpace does not create the game object for you.

So, do the following:

@create Sun
@create Mercury
@create Venus
@create Earth

And so on for all the objects you plan on inserting. You can do these one at a time if you so choose. Keep track of the DBREF's of each object as they will be needed for the next steps.

Now the objects need to be inserted into HSpace. To do this, the *@space/addobject* command will be used. *@space/addobject* requires two arguments: the thing to be inserted and the type of object to be added. The thing is the object created above and the type is the HSpace type identification number. See HS-TYPES for a current list of identification numbers. As we are adding planets, the type will be 3. Do the following to insert our new planets.

@space/addobject sun=3
@space/addobject mercury=3

Continue along for each new planet. To see a list of the current contents of HSpace, you can use *@space/list* again. Since planets are being added, it is useful to just list just that type by:

@space/list objects/0=3

This lists all objects in Universe ID 0 of type 3. By default, *@space/addobject* will insert new objects into Universe 0 at the X,Y,Z coordinates of 0,0,0. It also provides a default name for the new objects. Having everything globbed together at the center of the universe with no relevant names isn't likely what you want. To adjust positions, names, and universes, the *@space/set* command comes into play. *@space/set* is used to change the attributes associated with all HSpace objects. For the new planets, start by modifying the core attributes: name, coordinates, size and universe ID as appropriate. For example:

```
@space/set Earth/name=Earth
@space/set Earth/size=3
@space/set Earth/x=5000
@space/set Earth/y=9500
@space/set Earth/z=0
@space/set Earth/uid=0
```

The name is the name of the object as it appears in space. It can be different from the actual game object's name or it can be the same. Size is the size of the object in space from 1 to n where n is the maximum value chosen for your game. Size affects how far away sensors on space vessels can detect the object. X, Y, and Z are the three-dimensional coordinates of the object. You can decide the layout of your 3D space as appropriate to your game. Finally, the UID is the universe ID to which this object belongs. The UID is the number shown in *@space/list universes*. If you only have a single universe, 0, you do not need to change the UID of the object as it will be placed into that universe by default.   If you have multiple universes, this needs to be set appropriately for each object.

Repeat the above sets for each object you want in this first universe. You can verify the changes using the *@space/list objects/0=3* as noted above. As with all intensive building, you may want to save the game frequently with *@dump* to preserve changes in the event of a crash.   *@dump* automatically calls the save routines for HSpace as well as the game.

Planets can be inhabited or uninhabited. If a planet is to be inhabited, one or more landing locations should be added. Landing locations are described later in the document.

Asteroid Belts

Asteroid belts are created identically to the example of the planets above with a type ID of 7. Asteroids have one additional attribute: Density.   Density refers to the amount of space occupied by the asteroids in the area from 1 (default) to 100 percent.. Ships can be damaged when within the region encompassed by the asteroid belt. The region is determined by the size of the object * 100 centered around the object's 3D coordinates. Damage is randomly determined based on the density and size of the asteroid belt along with the speed of the target ship. You can see the current list of asteroid belts with *@space/list objects/0=7*.

An Example:

```
@create Kepler A
@space/addobject  Kepler A=7
@space/set Kepler A/size=5
@space/set Kepler A/density=20
@space/set Kepler A/x=23000
@space/set Kepler A/y=-4320
@space/set Kepler A/z=-7623
@space/set Kepler A/uid=0
```

Black Holes

Black holes are identified in HSpace as object type 5. They have only the standard attributes but have the side-effect of dragging space vessels into their gravitation field. A black hole will pull any ship within its size *100 into it causing damage to the vessel. Damage varies from 0 at the edge of the gravitational field to 1000*size at the center. Insertion is identical to planets with the exception of the type ID number being a 5.

Nebulae

A nebula is a collection of gas or dust in interstellar space. Nebulae have two additional attributes in addition to the standard set: DENSITY and SHIELDAFF. Density is identical to that of asteroids and defaults to 1. SHIELDAFF is a floating-point percentage value from 0.00 to 100.00. It determines how effective a ship's shields are while within 100*size of the nebula. Nebulae are inserted into HSpace just like planets but with a type ID of 6.

Wormholes

Wormholes are two black holes with an immense gravitation field between them. This field is so intense it folds the fabric of space and time, creating a passage from one point in space to another. Wormholes have a stability value, as a percentage, which represents the chance of successfully gating the wormhole. Failure to gate results in the destruction of the ship. Wormhole gating can move a vessel either from one point to another within a universe or between universes. Wormholes have the following attributes in addition to the standard ones.

> BASESTABILITY: The percentage around which the stability of the wormhole fluctuates.
> STABILITY: The current stability percentage for the wormhole.
> FLUCTUATION: The margin within which the stability fluctuates.
> DESTX: X coordinate of the destination
> DESTY: Y coordinate of the destination
> DESTZ: Z coordinate of the destination.
> DESTUID: The UID of the destination universe.

All additional attributes default to 0. For example if the basestability and stability of 50.0, a ship would have a 50% chance of successfully gating through the wormhole. If the fluctuation were 10.0, the stability would vary between 40-60% chance of success. The current stability value is determined randomly at the time the ship attempts to gate through the wormhole. Wormholes are represented in HSpace as type 4 when inserted with *@space/addobject*.

Celestial Objects Summary

All celestial objects are represented by both a game object created via *@create* and an HSpace Object inserted with *@space/addobject* and can be removed with *@space/delobject*. A type ID represents HSpace objects. This type is taken from the HS-TYPES list, when added into the space system. All Hspace objects have a core set of attributes that can be set. Some celestial object types have additional attributes that can be set as well. Attributes on HSpace objects are modified with *@space/set*. Once a game object is inserted into space, it should not be deleted unless *@space/delobject* is used prior to destroying it.

Building a Ship Class in HSpace by Kyle Forbes

What is a ship class?

A ship class could also be thought of as a template for ships you will build in the game. Let's just quickly imagine that each time you want to build a ship, you have to specify all of the information for that ship. Not only do you have to build the rooms, install the consoles, and tell the ship what it's name is, but you also have to tell it that it has engines, sensors, and other systems. You have to tell it that it has a hull of 500 points strength, two shields of 200 points each, and a cargo bay of 200 units. That's not all you would have to tell the ship about. For each ship built, there is a lot of information that the ship needs to know in order to function properly. That's where a ship class saves you a lot of time. You setup the ship class with the template information, and then you build ships of that class. Of course, you can build multiple ship classes. In fact, you can make hundreds of them.

Overriding Class Information

What's nice about HSpace, though, is that even though your ship is of a certain class, you can still "override" that class information and set variables at the level of the individual ship. Let's say, for example, that you have a ship that was built as an Avenger Class Destroyer. You want the owner of the ship to be able to install a more durable hull on this ship, but you want all of the other ship information to remain just as the Avenger Class specifies (shields, cargo size, etc). You can tell that individual ship that it has a hull value different from the ship class, and it will work! You'll see examples of how to do that later…

The Making of a Class

In previous versions of HSpace, putting a new class into the system was a piece of cake (I prefer cheesecake). Unfortunately as HSpace became more flexible, we lost a bit of convenience in constructing classes. In fact, it's probably the hardest thing to do in HSpace, which is why this guide is going to be your next best friend. Fortunately, you don't have to build classes that often. Perhaps you install a new class in the game once a day, though you may initially install many classes in your game. It is nothing compared to the many ships you may create each day, which is much easier than creating a class. To construct a class, you're going to use the following commands in HSpace:

- @space/newclass
- @space/addsysclass
- @space/setclass
- @space/syssetclass
- @space/sysinfoclass
- @space/dumpclass

Yes, there are six of them, and you'll be using them like a professional. You'll use these commands to 1) create a new class (newclass), 2) add engineering systems to the class (addsysclass), 3) set general information on the class (setclass), and 4) set information for systems on that class (syssetclass). The sysinfoclass and dumpclass do not set any information on the class. They simply allow you to retrieve current information about the class (dumpclass) and the systems attached to that class (sysinfoclass). You may abbreviate these commands so long as HSpace recognizes them as the unique commands that they be. For example, HSpace has an @space/set command to set information on HSpace objects. You can abbreviate @space/setclass as @space/setc, but you cannot abbreviate it as @space/set. HSpace will think you're referring to the @space/set command instead of @space/setclass.

Constructing a new ship class involves two primary steps. You must first create the class and set the general information on it. Then, you must add engineering systems (whichever you choose) to the class and set their information. You must add engineering systems because different types of ships may have different types of systems located on them. Rather than HSpace making all ships have the same systems and then you telling HSpace which are present or not, the software allows you to specify only the systems you want to be located on ships of a certain class.

Creating a New Class

Creating a new class is simple. You simply use the @space/newclass command. Its syntax is as follows:

```
@space/newclass <class name>
```

When you type this command, HSpace creates a new class with the next class ID available, starting with 0. If you have 10 classes, they will be 0 – 9, so the next will be 10. You can view your currently installed classes with the following command:

```
@space/list classes
```

Type the following command to setup a new test class:

```
@space/newclass Demo Class
```

Your new class should be created. To view your new class, type the following command.

```
@space/dumpclass 0
```

In this command, though, I'm assuming your new class is ID 0. If it is not, substitute 0 with whatever ID your class is. At the very least, you should have a class 0 at all times in HSpace (unless you have no classes). What you should see is the following:

```
Ship Class: 0   Demo Class
----------------------------------------------------------
Ship Size : 0                     Cargo Size: 0
Ship Crews: 0                     Max Hull  : 0
Can Drop  : NO
```

Setting General Class Attributes

As you see in the example above, there isn't much information that comes with a new class. The ship size is 0, which is non-existent to HSpace. At the very least you should have a ship size of 1 for the ship to even appear on sensors. The hull is empty, which makes the ship open to all sorts of nasty space diseases. You need to set this information on the ship, and that's easy. Try the following commands, which assume your class ID is 0. Again, if it is not, use the class ID of your Demo Class. For the rest of this document, I'll assume your class ID is 0 for this new class.

```
@space/setclass 0/size=1
@space/setclass 0/maxhull=100
@space/setclass 0/can drop=1
@space/setclass 0/cargo=10
```

In HSpace 4.0, the Ship Crews attribute is NOT used, so don't worry about it. The above commands set all of the information you need at the general level. Note that the variable to set the hull points is maxhull and not "max hull." For variables on ships and classes (and anywhere in HSpace) that require a YES or NO, use 1 for YES and 0 for NO. Now, type @space/dumpclass 0 again to see that your class information has changed! Now it's time to add systems …

Adding Systems to Classes

As mentioned previously, you need to add engineering systems to your classes so that new ships of your classes know what engineering systems they should have present during their operation. You can have any of the following engineering systems:

Internal Computer	Sensor Array
Engines	Life Support
Maneuv. Thrusters	Comm. Array
Fore Shield	Aft Shield
Port Shield	Starboard Shield
Jump Drive	Reactor
Fuel System	

Although ships may have any of these systems, that does not mean those systems will show up on the systems report of the engineering status on the ship. Some of these systems are marked as invisible so that players don't directly interface with them. For example, you may want a player to transfer power to and from engines, but you don't want a player directly interacting with the fuel system, which HSpace uses to allocate fuel to the reactor and engines.

To add an engineering system to the class, we'll go through the following example. Although I could show you how to add each and every system to the class, that would be incredibly redundant, and you'll probably pick this up really fast. We're going to work with the reactor of the ship, since that's one of the first things you'll always add to a new class. First, let's add the system to the class. Type the following command:

```
@space/addsysclass 0=reactor
```

Your reactor should now be added to the class, so let's see what sort of information it contains. Type the following:

```
@space/sysinfoclass 0=reactor
```

What you should see is something like the following:

```
---------------------------------------
Name     : Reactor
Visible  : NO
```

```
Damage   : 0
Tolerance: 0
Opt Power: 0
Cur Power: 0
Stress   : 0.0
MAX OUTPUT: 0
DESIRED OUTPUT: 0
CURRENT OUTPUT: 0
----------------------------------------
```

That's a lot of information, but what does it all mean? Each engineering system contains a base set of information, common to all of the engineering system types. This information includes the system name, its visibility to players working with the systems, its current damage, stress tolerance, optimal power, current power, and stress level. Additionally, each type of system may provide more variables that can be queried and set for that particular system. In the case of the reactor we're using, the MAX OUTPUT, DESIRED OUTPUT, and CURRENT OUTPUT are all specific to the reactor system.

You will not set all of these variables on the system for the class. For example, you'll want to set the MAX OUTPUT of the reactor, but you don't want to set the CURRENT OUTPUT. Why is it there? It is there because you are adding complete engineering systems to the class, but only as a template for ships that will have their own engineering systems. Wow, what does that mean in English? It means that you are just providing basic system information for the ships of that class. Those ships, then, will have reactors that have maximum outputs equal to the maximum output of the reactor of the ship class. Additionally, the desired output of the reactor will be set on the reactor for each individual ship. You don't want to tell all ships of your class that the desired reactor output is something other than 0. That's up the players to specify how much power they want their reactor to currently output. Thus, we're only going to set the following variables:

• Tolerance	-	The stress tolerance of the system (1 .. n)
• Max Output	-	The maximum output of the reactor.

Here's how it's done, watch closely!

```
@space/syssetclass 0:reactor/tolerance=1
@space/syssetclass 0:reactor/max output=50
```

Why don't we set the optimal power? That's easy to answer. You will typically set the optimal power for other engineering systems. However, the reactor is a bit different. It does not consume power – it produces it! Thus, the max output is the variable that indicates how much power the reactor can produce (without being overloaded). Normally you would specify the optimal power for a system to indicate the maximum power it can be allocated without being overloaded.

Now, if you type @space/sysinfoc 0=reactor, you'll see the following:

```
----------------------------------------
Name     : Reactor
Visible  : NO
Damage   : 0
Tolerance: 1
Opt Power: 0
Cur Power: 0
Stress   : 0.0
MAX OUTPUT: 50
DESIRED OUTPUT: 0
CURRENT OUTPUT: 0
----------------------------------------
```

That's all you need to do for the reactor! If you follow these same steps for the other systems you want to add, you'll have a fully functioning ship in no time. All you need to do to create a ship of this class is to build your ship, create an object that represents your ship, and type the following command:

```
@space/activate <object>=0
```

Again, I'm assuming 0 is your class ID.

There's one more thing I want to show you before we finish this section of the discussion. Type the following command again:

```
@space/dumpclass 0
```

Look how it's changed! It shows there's an engineering system now – the reactor. See if you can add the rest of the engineering systems and set their attributes. Try adding shields – they have lots of variables to be set.

What the Variables Mean

Although each engineering system has its own set of variables that need to be set, each engineering system shares a base set of variables that you will need to set. This has already been mentioned, so let's get on with what these variables are:

Tolerance
The tolerance variable indicates how quickly the system stresses when overloaded. There is no set range for this variable, though it has to be a minimum of 1. If you set this variable to 0, then any overloading of the system creates instantaneous stress at the level of 100%. As the stress on a system increases from 0 to 100%, the chance that the system will incur damage increases. The higher you set the tolerance variable, the more resistant it is to stress.

Damage
The damage variable does not need to be set on the systems at the class level, though it is important to know what it does at the ship level. Damage values can range from 0 to 4, indicating no, light, medium, heavy, and inoperable damage.

Opt Power
To set this variable at the class level, you need to specify the "optimal power" setting, not the opt power variable (@space/syssetc 0:system/optimal power). The optimal power is the maximum power that can be allocated to the system without overloading and stressing it. Any power settings less than the optimal power are considered underloading the system, and any power settings over the optimal power are considered overloading. Overloading a system stresses the system, while underloading the system destresses it. It is important to note that stressing a system and then setting it to 100% power allocation only slowly destresses the system. The fastest way to reduce system stress is to shut the system down.

Cur Power
Cur power, like opt power, actually stands for current power, which is the name that should be used when setting this variable. It does not need to be set at the class level, but it indicates at the ship level the amount of power currently allocated to the system.

Stress

The stress variable does not need to be set at the class level. At the ship level it indicates how much stress the system has currently taken on a level from 0 – 100% stress. At 100% stress the system will likely be inoperably damaged in short time.

Miscellaneous Information

While I’ve covered the basics of ship class creation in this short guide, there are other things to consider. With each release of HSpace, more systems will probably be available, and each of those systems will serve some purpose on a ship. You may choose to add these systems to your classes or leave them out, but some are quite necessary. A ship with no reactor will clearly have no power to allocate to its systems. Without power, those systems will not perform. You can add everything to a ship but engines, but without engines that ship is not going to move.

You can, however, simulate unlimited fuel on a ship by leaving the fuel system off of the ship class. Without a fuel system, the ship does still function, and it consumes no fuel.

You may add up to four shields to a ship class, but you must add them in pairs. A ship class can have 0, 2, or 4 shields, but not 1 or 3. The first two shields added are always front and rear, even if you add the port and starboard instead of the fore and aft shields. The next two shields are in the port and starboard locations. It does not matter if you add port before aft or starboard before fore. The port shield is always the port, the aft is always the aft.

Building Your First Ship

This section of the document assumes you have already setup a universe to contain space objects and have setup a class utilizing the previous section. If you have not already done so, please take a few moments to re-read the previous sections and add those elements to your game.

Activating a Ship

A ship in HSpace is an instantiation of a ship class. Upon activation, the ship is set with all the values contained in the class. However, the ship can be customized further if desired by utilizing various @space commands to modify its abilities and systems. Ships are nothing more than a simple object but with the proper setup they can contain rooms used to describe the cockpit, engineering room, or even the loading bay if the ship is a cargo hauler. Massive ships representing space stations can be constructed with many rooms allowing areas for role play. However, let's start simply by using the simple demonstration class setup earlier.

The first thing that is needed is an object to represent the ship. So create it.

@create Demo Ship I

You may wish to jot down the DBREF of the object created as it can be used in place of the actual name of the ship for any of the following steps. Next, HSpace needs to know that this object is a ship. This is accomplished by activating the ship. Activation sets the ship with the default values from the specified class.

@space/activate Demo Ship I=<class number>

If you are reading along through this document, you likely only have a single class. If you have multiple classes, use @space/list classes to select the class and use the appropriate class number. Once the command is complete, you can type *@space* and see that a new ship has been added. If you have multiple universes, you will need to change the ship to the proper universe id via:

@space/set Demo Ship I/uid=<UID>

You might also want to set the ship's name within HSpace so it can be used in relaying messages to other ships. This is done by:

@space/set Demo Ship I/name=<newname>

You may also want to set the X, Y and Z coordinates for the ship using the same command. E.g. @space/set <ship>/<X|Y|Z>=<coordinate>. @space/set demo/x=1000, @space/set demo/z=-10000

Adding Rooms to the Ship

For our purposes, let's assume that Demo Ship I is just a small fighter with a single area for occupants. To make the ship more pleasant, a room will be added that can be used for this purpose. Such rooms are normal MUSH rooms that have been added to the ship via the @space/addsroom command. These rooms are not linked to the normal MUSH grid and may need to have the floating flag or various warnings removed from them depending on the type of server selected. So setup the room:

@dig Demo I Cockpit
@describe Demo I Cockpit=Two seats dominate the fore of the craft used to house the pilot and one passenger. A wide assortment of lights, gauges and levers surround the front seat.

Now, add the room onto the ship object.

@space/addsroom Demo Ship I=<dbref of Demo I Cockpit>

Now our ship has a room. If you are building a larger ship with multiple rooms, you can dig and describe the entire ship setup prior to adding the rooms to the ship. Once the ship is complete, you can walk through the entire area adding each room to the ship in turn.   Exits leading in between rooms do not need to be handled in anyway. They are merely normal exits just like you would have in a normal gaming area.

Setting Up Consoles

Unless you plan on typing a whole lot of @space commands, it is now time to add consoles to the ship. Consoles are MUSH objects with $commands set on them that interface to the HSpace system. They are typically configured as parent objects. You can find example consoles in the HSpace Archive at http://www.moosh.net/pub/mush/hspace/ if you do not already have them. Consoles also need to be set with the proper flags and have the proper permissions. (!no_command for Penn, Commands for Tiny and inherit or be set with Wizard permissions). Once you have a console uploaded, you can proceed with adding a console to our demo ship. I chose a single parent console that combines the majority of the @nav and @eng commands.

Setup the console within the MUSH:

@create Console
@parent Console=<parent console dbref>
@set Console=!no_command (Penn) or Commands (Tiny)
@set Console=Inherit

Add the console to our demo ship:

@space/addconsole <ship dbref>=Console

Finally, place the console in the cockpit of the ship.   You should be able to man the console and startup the ship utilizing the commands as set on the parent console. As those commands can vary based on the console parent, they are not covered here. Larger ships may have more than one console. Often, there is a console for the navigation commands, one for engineering, one or more weapons consoles and even command consoles.   To HSpace, a console is just an object that is allowed to manipulate the ship. The softcoded commands on (or inherited from a parent), allow HSpace administrators to tailor console functionality to what is desired for the specific console interface.

Boarding and Disembarking

In order to board to depart your ship, HSpace needs to have a couple more things set.   First, to get off the ship, you must set the SHIP attribute on the room(s) that can be used for disembarking.   Since Demo Ship I only has a single room, the choice is clear.

&SHIP <cockpit room dbref>=<dbref of ship object>

Once that is set, you should be able to use the disembark command to leave the ship. However, HSpace also needs to be told where people boarding the ship end up. To specify this, set the BAY attribute on the ship object with the destination room for boarding. Again, the Demo Ship is simple as it only has one room.

&BAY <dbref of ship object>=<cockpit room dbref>

Most ships only have a single room for both of these. In fact, HSpace only handles a single location for boarding but you could have multiple locations for disembarking. It doesn’t really make sense to have a 1-way exit in a spaceship though.

That’s it. You should now be able to fly around the HSpace universe. If the ship class you used has a fuel system, you will need to fill it up using @space/sysset <dbref>:fuel system/burnable fuel=<amount> and reactable fuel as well. So if it doesn’t power up properly, make sure to check for that.

Communication Link Setup for HSpace

Personal Communications

HSpace provides the ability for game administrators to setup personal communication systems. These are often referred to as Communication Links or Commlinks. Often, these are man portable devices that are carried by players to communicate between themselves. Beware, this process is not used for setting up ship communications. Ship objects utilize a different setup. However, the commlinks can be used for listening or transmitting to ships as well as to other communication links.

Basic Setup

Communication links require three core elements. First, they must be set with the HSPACE_COMM flag. This flag tells the HSpace engine that the object is a communication device and should be checked for reception of messages. Next, the object must have a COMM_FRQS attribute set on it containing a space separated list of frequencies that are being monitored. Again, this is used by HSpace to determine if the object should be evaluated for reception of a given message. Finally, the object needs to have the COMM_HANDLER attribute set. This attribute is evaluated after HSpace has determined the object has the HSPACE_COMM flag, it is within range of the sender of the message, and the object is listening on the appropriate frequency. The COMM_HANDLER is passed three arguments: %0-the message, %1-the frequency, and %2-the dbref of the sending device. Think of the COMM_HANDLER as being evaluated like a normal u() call. E.G., [u(comm._handler,%0,%1,%2)]. Typically, the COMM_HANDLER make use of some function to pass the message along to its location.

Sholevi's Quickstart Guide

Here is a brief and simple tutorial, or quickstart guide, to getting two hspace comm devices communicating with each other:

1. Create two devices. Set the HSPACE_COMM flag on both of them.
2. Set attributes COMM_FRQS on both of them to: 101.01. This let's them LISTEN on this channel. You can add more channels by space-separating them, but this is just for testing for now.
3. Set attributes on both of them COMM_BROADCAST to: 101.01. This is not required by HSpace, but you will want to have something like this on your parent when you set it up. This indicates the channel that the unit will speak on.
4. Set attributes on both devices ACTIVE to this: 1. This is also not required, but you will want something like it on your parent, to indicate whether the unit is on or off.
5. Set attributes on both devices COMM_HANDLER to this:

   Code:
   ```
   if(v(active),pemit(loc(me),Message from
   [name(pmatch(loc(%2)))] on channel %1:  %0))
   ```

6. Set a command on each object CMD_COM to this:

   Code:

```
$com *:@pemit
%#=if(match(hs_comm_msg(1,1,1,1,1,5,v(COMM_BROADCAST),%0),F
ailed),You message could not be sent.,You sent your message
to channel [v(comm_broadcast)].)
```

In the hs_comm_msg() of that command, the parameters are, respectively, the source universe, the destination universe, the x, y and z of the sender, the range of the message (in hspace units), the channel, and the message. Mind you, I'm just typing this up to give you an idea of how it works, and I haven't tested it, so don't be surprised if it's not perfect. To send a message, the holder of the device would just type: "com Hi".

At this point, you have all that you need to test that the system works. As long as both the sender and receiver are in universe 1 and are within the range, you could send a message, which would automatically call the COMM_HANDLER of the other device as a function.

Now, to recap, repeat these steps for each device to test the HSpace comm system. Once you have it working and passing the test, you can devise a more complicated Parent for your communications system, and parent the devices to it.

Using HSPACE_LOCATION

As you can see in Sholevi's example of the com command, he's hard-coded the source and destination UIDs, as well as the x, y, z coordinates for the transmission via HS_COMM_MSG(). Luckily, HSpace provides a way to automatically retrieve that information by looking on the transmitting player for the HSPACE_LOCATION attribute. This attribute contains the dbref of the current space location that person is in (Note: this will not be set until the person interacts with the space system so you may need to manually set it in chargen or upon creation of the player). Utilizing that dbref, you can get the current UID, X, Y and Z information for that object using HS_GET_ATTR(). Making use of the HSPACE_LOCATION requires the communication device has sufficient powers to read attributes from any object such as wizard flag or inheritance or the see_all power.

Thus, the code could become:

```
$com *:@pemit
%#=[setq(0,get(%#/hspace_location))][if(match(hs_comm_msg(hs_get_attr(%
q0,uid),hs_get_attr(%q0,uid),hs_get_attr(%q0,x),hs_get_attr(%q0,y),hs_g
et_attr(%q0,z),5,v(COMM_BROADCAST),%0),Failed),You message could not be
sent.,You sent your message to channel [v(comm_broadcast)].)]
```

This setup allows the unit to dynamically get the origin location from the sender and utilize that information in the message transmission. For the range, you could have a setup that allows the range to be dynamically set by the unit and thereby replace the last hard-coded value of 5 with a user-settable value.