Habitat Ephemera

Where Stuff Is

The following is an overview of the various Habitat directories. The detailed information about each chunk of stuff will be given later.

The Habitat source that we have developed is kept on the Quantum Stratus system under the user account 'guest.lucas'. The important elements of the directory hierarchy are:

```
#d010>lucas
        microcosm
                Actions
                Classes
                Ghu
                Grabthese
                Linkable
                Misc
                Structs
        toolbox
```

#d010>lucas is the account's home directory. Not much of interest is here except for our abbreviations file, which is highly non-standard by Quantum's reckonning (we have tried to emulate the Unix environment as much as possible, rather than to use the Quantum Stratus abbreviation set; this causes no end of trouble when Quantum folks try to do things while logged in as us; if we had it to do over again...)

#d010>microcosm is the master source directory for Habitat. The name is historical, since Habitat used to be called "MicroCosm". In this directory are

1) various include files that are used by our code
2) directories for the rest of the source
3) command files and other miscellaneous working junk files

#d010>microcosm>Actions contains the source for the generic action routines. These are object behaviors which are shared among multiple object classes.

#d010>microcosm>Classes contains the source for the various object classes.

#d010>microcosm>Ghu is the working directory for Ghu development and maintainance.

#d010>microcosm>Grabthese is where we place updated source files for Janet to grab them and incorporate them into the production system.

#d010>microcosm>Linkable is where all the compiled object files for all the Habitat code lives (all the code that we maintain, that is).

#d010>microcosm>Misc contains miscellaneous other Habitat source files that defy categorization.

#d010>microcosm>Structs contains '.incl.pl1' files for the various different object class structs.

#d010>toolbox has various utilities that we use. The most notable thing to be found here is the release version of Ghu, but it also has a number of other minor utilities which we have found useful.

Overall Structure of the Host System

The Habitat system is built around an "object oriented" design philosophy. The entire Habitat world is a collection of "objects". Abstractly, an object is a set of code and data that together act like some concrete entity. For example, regions, avatars, vending machines, magic wands, heads and teleport booths are all different kinds of objects. Ideally, we would like some sort of programming environment/operating system that incorporated this object model into its fundamental basis for operations. However, the world is not ideal, so we have approximated an object-oriented system with a conventional system that has a sort of object-oriented form.

The host system code can be divided into three very broad catagories.

The first could be called the "skeleton". This is the framework on top of which the Habitat system runs. The skeleton consists of the main code of the 'regionproc'; the various external Habitat processes such as the database process, the hatchery, and so on; and the non-Habitat host code that forms the rest of the Q-Link system. All of this code was provided by Quantum and we will not describe it here except in the most summary outlines (though we will make reference to it where appropriate).

The second code category we'll call the "objects". This is the collection of routines which implement the definition and behaviors of the various different objects that make up the Habitat world. These follow a rather rigid format and can be described as a group. Along with the objects are a few sets of special code that handle major sub-systems. Notable among these are the help system, the magic system, and the curse system. In the interest of narrative clarity, these will be described in sections of their own.

The third category of code could be called the "helpers". These are various routines we have written to make the world work more smoothly for us. They are not, as a rule, essential to the operation of the system, but understanding them is essential to understanding the system. Fortunately, their job is to make things simpler rather than more complex.

Summary of Operation

The following section is a summary of the operation of the main body of the Habitat system. This is all described elsewhere in the early design documents, but we'll repeat it here in broad outline in order to provide some context for what follows.

The Habitat world consists of a set of regions, each of which is a place you can visit with your Avatar. Each region contains some number of objects. Each object belongs to a particular "class". The class determines, in effect, what sort of object it is: how it is to behave and how the data that describes its state are to be interpreted. In principle there are 256 possible object classes, numbered from 0 to 255. In practice only a little more than half of these are actually used. Each object is represented by a data record that is kept in the host's memory when the object is in an "active" region and in a disk database when the object is not in an active region. A region is said to be active when there are one or more online users (avatars or ghosts) in it.

The process of activating a region when the first user enters consists of reading the object records into memory from the appropriate databases ("databases" is plural because regions and avatars, being special in various ways, are kept in separate databases from the other objects). Deactivating a region consists of writing the records back out again when the last user leaves (for efficiency, we only bother to actually write those records which have changed in the interim). Each object in an active region is assigned a temporary identifier called a "noid" (short for Numeric Object IDentifier) which is a number in the range from 0 to 255. The region itself is represented by an object which always has noid 0. The regionproc keeps an internal table that maps from noids to the object records themselves.

Each transaction between the C64 and the host is couched in terms of the object model. Each message from the C64 is directed to a particular object in the region that the C64's user finds himself in. The first byte of the message, in fact (after stripping off the various telecommunications protocol bytes), is the noid of the object to which the message is addressed. The byte after that is a number that indicates what the C64 is requesting of the object. The remaining bytes, if any, are request-specific parameter information.

When a message arrives, the regionproc extracts the noid from the message and uses this to locate the record corresponding to the object itself. This record contains, among other things, the class number of the object. The class number is used as an index into another table kept by the regionproc that contains the "class definitions". Part of a class's definition is an array of pointers to procedure entry points that correspond to the class's various "behaviors". The request number (the second byte of the message) is used as an index into this array, and the indicated procedure is called. This procedure then carries out whatever action is appropriate for the given request, including transmitting a response message to the C64 if that is appropriate. Before calling the behavior procedure, the regionproc also sets up a number of standard pointers and global variables that the behavior can look at to find out about the environment in which it is executing.

All of the above machinations, with the exception of the behavior routines themselves, are performed by the "skeleton" code mentioned earlier. This is the code that was developed as Quantum's portion of the project.

The Objects and Class Definition

Each object is defined by two PL/1 source files. The first defines the procedures to initialize the object and implement the object's various behaviors, if any. The second is an include file that defines a PL/1 structure that describes the object's state information (i.e., the contents of the object's database record *after* having been read into memory).

The first file is called the "class" file and lives in the 'Classes' directory. The second is the "struct" file and lives in the 'Structs' directory. For a given class, say "foo", we would have the files 'Classes>class_foo.pl1' and 'Structs>struct_foo.incl.pl1'.

The class file defines a procedure 'initialize_class_foo' that is called by the regionproc at system startup time. This procedure sets up the class table entry for this class so that it will work right when the time comes. This file also contains the definitions of any class-specific behavior procedures that may be required for the object. For example, here is a very simple class file which defines the 'ball' class:

```
/*
 *   class_ball.pl1
 *
 *   Ball object behavior module for Habitat.
```

```
 *
 *   Chip Morningstar
 *   Lucasfilm Ltd.
 *   9-April-1986
 */

%include 'microcosm.incl.pl1';
%include 'defs_action.incl.pl1';

initialize_class_ball: procedure;

     %replace BALL_REQUESTS by 3;

     declare a(0:BALL_REQUESTS) entry based;
     declare class_ball_actions pointer;
     declare 1 ball based %include struct_ball;

     %replace I by CLASS_BALL;

     Class_Table(I).capacity = 0;
     Class_Table(I).max_requests = BALL_REQUESTS;
     Class_Table(I).alloc_size = size(ball);
     Class_Table(I).pc_state_bytes = 0;
     Class_Table(I).known = true;
     Class_Table(I).opaque_container = false;
     Class_Table(I).filler = false;

     allocate a set(class_ball_actions);
     Class_Table(I).actions = class_ball_actions;

     Class_Table(I).actions->a(HELP)  = generic_HELP;  /* 0 */
     Class_Table(I).actions->a(GET)   = generic_GET;   /* 1 */
     Class_Table(I).actions->a(PUT)   = generic_PUT;   /* 2 */
     Class_Table(I).actions->a(THROW) = generic_THROW; /* 3 */

end initialize_class_ball;
```

The following notes apply:

'microcosm.incl.pl1' is the general purpose Habitat include file. It declares all the basic global variables, types and constants that are used throughout the system. It should be included in just about everything.

'defs_action.incl.pl1' declares the entry points for the generic behavior routines found in the various files in the 'Actions' directory. Note that many objects (this one among them) share common behaviors. For example, the code to put down or pick up an object is, with rare exceptions, the same regardless of the object's class. Thus, we have generic routines to handle the GET and PUT requests, instead of implementing these procedures anew in each object class.

'initialize_class_ball' is the required initialization routine for this class. It is called by the regionproc at system startup time. All classes MUST have a routine of this sort.

'BALL_REQUESTS' is the maximum request number that objects of class ball will be expected to recieve. It is defined here as a convenient constant that you will see reference to in several places in the routine.

'struct_ball' is a string constant that is defined by the include file 'defs_struct.incl.pl1' which is in turn included automatically by the include file 'microcosm.incl.pl1'. This string constant expands to the file name for the "struct" file, mentioned above. The 'defs_struct.incl.pl1' include file defines one of these string constants for each class. The idiom

```
        declare 1 foo based %include struct_foo;
```

is a common one that will be seen again and again throughout the Habitat code.

'Class_Table' is a global table that is maintained by the regionproc. The primary purpose of this init procedure is to fill in the Class_Table entry for this class. This consists of assigning various properties and allocating and setting up the array of pointers to the behaviors.

'capacity' is the maximum number of objects that objects of this class may contain. For objects which are not containers (e.g., this one), this should be set to 0.

'max_requests' is the maximum request number that objects of this class will accept. If the regionproc recieves a request to an object of this class that is greater than this number, it will drop the request on the floor and put a diagnostic message in the run-time log file (i.e., this should never happen).

'alloc_size' is the amount of memory to allocate for objects of this class when they are read from the object database at region activation time.

'pc_state_bytes' is the number of bytes of data from the in-memory record to send to the C64 when someone sees an object of this class (the number of bytes in addition to the 6 which are sent for every object regardless of class). note that objects may have state information on the host which is not revealed to the C64.

'known' is simply a flag that says, "Yes, this class exists". This is used in the course of various diagnostics.

'opaque_container' is a flag that is set to 'true' if and only if the object is an opaque container, i.e., a container whose contents are not visible without explicitly looking inside it. This is 'false' if the object is either not a container at all (the case here) or if the object is "transparent" (e.g., a table).

'actions' is the array of pointers to the behavior procedures for this class. Note that it must be allocated dynamically since its size may vary depending on how many behaviors the class has.

The particular elements of the 'actions' array correspond to the various requests that the object will respond to. The request numbers are defined in the include file 'defs_message.incl.pl1' which is included automatically by 'microcosm.incl.pl1'. These requests are always interpreted relative to the object class (e.g., request 5 for class A does not mean the same thing as request 5 for class B). However, for the sake of consistency and diagnostics, we DO enforce the following conventions:

```
        request 0 is always HELP
        request 1 is always GET
        request 2 is always PUT
        request 3 is always THROW
```

if the class in question does not respond to one of these requests, it should set the corresponding array entry to 'illegal'. In the case of the ball object, ALL of these requests are handled by the generic behaviors, and so there are no ball-specific behaviors defined here.

The struct file for this example looks like this:

```
/*
 *   struct_ball.incl.pl1
 *
 *   Struct stub for ball instance descriptor.
 *
 *   Chip Morningstar
 *   Lucasfilm Ltd.
 *   9-April-1986
 *
 */
,    2    common_head         like instance_head
; /* terminates struct header from include file */
```

This is a trivial struct file, since the class ball has no state information that is peculiar to the class. It merely has the common information that all objects have which is defined by the struct ’instance_head’ in the ’microcosm.incl.pl1’ file. For the record, it is:

```
/*
 *   instance_head.def.incl.pl1
 *
 *   The common header shared by ALL object instance descriptors.
 *
 *   Chip Morningstar
 *   Lucasfilm Ltd.
 *   9-April-1986
 *
 */
declare 1 instance_head       based,
          2    avatarslot     binary(15),
          2    obj_id         binary(31),
          2    noid           binary(15),
          2    class          binary(15),
          2    style          binary(15),
          2    x              binary(15),
          2    y              binary(15),
          2    position       binary(15),
          2    orientation    binary(15),
          2    gr_state       binary(15),
          2    container      binary(15),
          2    gr_width       binary(15),
          2    gen_flags(32)  bit(1);
```

The meanings of the various fields are described elsewhere.

An example of a slightly less trivial class definition is the pawn machine. Note the similarities of form with the definition of class ball:

```
/*
 *   class_pawn_machine.pl1
 *
 *   Behavior module for object class pawn_machine.
 *
 *   Chip Morningstar
 *   Lucasfilm Ltd.
```

```
 *   6-October-1986
 */

%replace PAWN_MACHINE_CAPACITY by 1;

%include 'microcosm.incl.pl1';
%include 'defs_helper.incl.pl1';
%include 'defs_action.incl.pl1';

initialize_class_pawn_machine: procedure;

     %replace PAWN_MACHINE_REQUESTS by 6;

     declare a(0:PAWN_MACHINE_REQUESTS) entry based;
     declare class_pawn_machine_actions pointer;
     declare 1 pawn_machine based %include struct_pawn_machine;

     %replace I by CLASS_PAWN_MACHINE;

     Class_Table(I).capacity = PAWN_MACHINE_CAPACITY;
     Class_Table(I).max_requests = PAWN_MACHINE_REQUESTS;
     Class_Table(I).alloc_size = size(pawn_machine);
     Class_Table(I).pc_state_bytes = 3;
     Class_Table(I).known = true;
     Class_Table(I).opaque_container = true;
     Class_Table(I).filler = false;

     allocate a set(class_pawn_machine_actions);
     Class_Table(I).actions = class_pawn_machine_actions;

     Class_Table(I).actions->a(HELP)  = generic_HELP;      /* 0 */
     Class_Table(I).actions->a(1)     = illegal;           /* 1 */
     Class_Table(I).actions->a(2)     = illegal;           /* 2 */
     Class_Table(I).actions->a(3)     = illegal;           /* 3 */
     Class_Table(I).actions->a(4)     = illegal;           /* 4 */
     Class_Table(I).actions->a(5)     = illegal;           /* 5 */
     Class_Table(I).actions->a(MUNCH) = pawn_machine_MUNCH;/* 6 */
end initialize_class_pawn_machine;

pawn_machine_MUNCH: procedure;
     declare 1 self based(selfptr) %include struct_pawn_machine;

     if (adjacent(selfptr) & self.contents->c(0) ^= NULL) then do;
          if (pay_to(avatarptr, item_value(ObjList(self.contents->c(0))))) then
do;
               call n_msg_1(selfptr, MUNCH$, avatar.noid);
               call n_msg_1(null(), GOAWAY_$, self.contents->c(0));
               call destroy_contents(selfptr);
               call r_msg_1(TRUE);
               return;
          end;
          call r_msg_1(BOING_FAILURE);
          return;
     end;
     call r_msg_1(FALSE);
end pawn_machine_MUNCH;
```

The following points are worthy of mention:

'defs_helper.incl.pl1' is an include file that declares a variety of "helper" routines that a behavior can call to perform various services. These will be discussed in greater detail below.

'capacity' is set to 'PAWN_MACHINE_CAPACITY', a constant that has no counterpart in class ball. This is because the pawn machine is container (capable of holding 1 object) and the ball is not.

Note that the actions array has a number of 'illegal' entries, since the pawn machine is not a mobile object (i.e., it cannot be picked up and carried).

The pawn machine has one behavior of its own, which is defined here. Notice the naming convention used for behavior routines: 'classname_REQUEST'. Generic behaviors (those corresponding to more than one class) have names of the form 'generic_REQUEST'.

The behavior itself contains many items worth discussing, but we will cover them in the following section when we explain the execution environment that behavior procedures live in.

The pawn machine's struct file looks like this:

```
/*
 *   struct_pawn_machine.incl.pl1
 *
 *   Struct stub for pawn_machine instance descriptor.
 *
 *   Chip Morningstar
 *   Lucasfilm Ltd.
 *   6-October-1986
 *
 */
,    2    common_head         like instance_head,
     2    contents            pointer,
     2    class_specific      ,
           3   open_flags     binary(15),
           3   key_hi         binary(15),
           3   key_lo         binary(15);
```

The 'contents' field appears only in those objects which are containers. It is filled in automagically by the regionproc when the container is opened. This class has class specific fields which are defined here. The fields shown here are required for any container, though in the case of the pawn machine they are a formality, since it can never be opened or closed, locked or unlocked, by a player. (These container-specific fields will be discussed later in more detail).

The Behavior Execution Environment

In addition to a variety of "helper" routines, which will be discussed in the next section, there are a number of important elements in a behavior procedure's execution environment that require explanation. Most of these are global variables that are declared by the include file 'microcosm.incl.pl1' and set by the regionproc before the behavior is called.

```
        %replace THE_REGION by 0;
```

This is (always) the noid of the region object in this region.

A series of string constants of the form 'struct_thingname' are defined to enable easy declaration of common data types. This was discussed in greater detail above.

```
        declare 1 o based %include struct_gen_object;
```

'struct_gen_object' is a generic object header that can be used to refer to the common state information of all objects. The based type 'o' lets us access such information with minimal fuss.

```
        declare 1 u based %include struct_user;
```

similarly, there is a "user struct" that contains user information that is pointed to by a field of avatar objects. This based struct lets us access its fields and thus do rare but sometimes necessary things that require access to the user's queue. This struct looks like:

```
/*
 *   struct_user.incl.pl1
 *
 *   Struct stub for UserList structure.
 *
 *   Chip Morningstar
 *   Lucasfilm Ltd.
 *   9-April-1986
 *
 */
,    2    U_Name              character(10) varying,
     2    U_Id                binary(31),
     2    U_Q_Id              binary(31),
     2    U_Q                 pointer,
     2    U_version           binary(15),
     2    object_slot         binary(15),
     2    esp                 ,
          3 to_uid            binary(31),
          3 to_qid            binary(31),
          3 que               pointer,
          3 lines             binary(15),
     2    last_mail_ts        binary(31),
     2    auto_destination    binary(31),
     2    auto_mode           binary(31),
     2    flags               ,
          3    U_mail              bit(1),
          3    cr_pending          bit(1),
          3    online              bit(1),
          3    incoming            bit(1),
          3    new_session         bit(1),
          3    ck_last_login       bit(1),
          3    filler              bit(10);
```

The useful fields are typically 'U_Name', 'U_Id', and 'online'. The user structs may be accessed via

```
        declare UserList(UsersPerRegion) pointer;
```

which points to the various users. You can find out the index into the UserList for a particular avatar via the avatar object's 'avatarslot' field.

You can map a noid to a pointer to an object via the global ObjList:

```
        declare ObjList(0:255) pointer;
```

entries in this list are index by noid and will be 'null()' if there is no object corresponding to a given noid..

```
        declare c(0:255) binary(15) based;
```

is declared so we can access the contents of a container object. Container objects always have a 'contents' field which is simply a pointer to an array of this form. Thus, if 'foo' is a container object, we can refer to, say, the third item in 'foo' as:

```
        foo.contents->c(2)
```

note that this value is a noid, not an object pointer. If there is no object in the particular container slot, this value will be NULL (i.e., 0). To get a pointer to the object itself you would have to say

```
        ObjList(foo.contents->c(2))
```

being careful, of course, to make sure that the object exists (i.e., that the pointer from the ObjList is not 'null()') before you try to do anything with it.

```
        declare avatarptr pointer external;
        declare 1 avatar based(avatarptr) %include struct_avatar;
        declare selfptr pointer external;
        declare 1 self based(selfptr) %include struct_gen_object;
```

Before any behavior is executed, 'avatarptr' (and thus 'avatar') is set to point to the object record for the object corresponding to the avatar whose C64 issued the request that we are processing. (Warning: this will be invalid if the user is a ghost.) Similarly, 'selfptr' (and thus 'self') is set to point to the object to which this request was sent. By this means any object can refer to itself as 'self' in its behavior code. Often, this declaration of 'self' is overridden by behaviors in order to make selfptr point to a different type of struct than 'struct_gen_object' (e.g., a flashlight behavior would want 'selfptr' to be declared as pointing to a 'struct_flashlight').

```
        declare request_string character(646) varying external;
        declare request(258) character(1) defined(request_string);
        %replace FIRST by 3;
        %replace SECOND by 4;
        %replace THIRD by 5;
        %replace FOURTH by 6;
        %replace FIFTH by 7
```

Before executing the behavior, 'request_string' is assigned the request message itself (after the telecommunications protocol information is stripped off). 'request' lets us individually index the bytes of the request. 'FIRST', 'SECOND', etc. are defined so that we can neatly refer to the parameters of the request (remember that the first byte is the noid to which the request is addressed and the second byte is the request number). Thus, the second parameter byte of a request would be

```
        request(SECOND)
```

often (usually, in fact) you will want the byte itself as a number, not as a character, so you will frequently see the idiom

```
        rank(request(SECOND))
```

which simply gets the byte as an integer.

The region itself is not represented on the host as an object, unfortunately. Information about the region is found in various globals. Notable are:

```
        declare Region binary(31);
```

The current region number.

```
        declare Region_name character(20);
```

The current region name.

```
        declare total_ghosts binary(15);
```

The number of ghosts in the region, and the notable sub-struct:

```
        declare 2  current_region,
             3 lighting                binary(15),
             3 depth                   binary(15),
             3 neighbor(4)             binary(31),
             3 exit_type(4)            binary(15),
             3 restriction(4)          bit(1),
             3 nitty_bits(28)          bit(1),
             3 max_avatars             binary(15),
             3 owner                   binary(31),
             3 entry_proc              binary(15),
             3 exit_proc               binary(15),
             3 class_group             binary(15),
             3 orientation             binary(15),
             3 object_count            binary(15),
             3 space_usage             binary(15),
             3 town_dir                character(1),
             3 port_dir                character(1);
```

full of all kinds of useful information. The global

```
        declare DayNight binary(15) external init(0);
```

contains the global illumination level, which controls whether it is day or night. For the time being it is always day, but this may change in the future.

Avatar, region and object records each have arrays of general purpose bit-flags called ’nitty_bits’. (Actually, avatars and objects have two sets of flags, ’nitty_bits’ associated with the object or avatar record and ’general_flags’ associated with the ’instance_head’ struct.) These bits are available for general use as needed. However, some are already allocated and you should avoid stepping on them:

```
        /* instance_head general flag constants */
        %replace RESTRICTED by 1;
        %replace MODIFIED   by 2;
```

The RESTRICTED bit means that the object can’t be taken out of a restricted region exit. The MODIFIED bit means that the object has been changed and should be written to the database (it is interrogated when the region is deactivated).

```
        /* region nitty_bits constants */
```

```
        %replace WEAPONS_FREE by 1;
        %replace STEAL_FREE by 2;
```

These make a region weapons free or theft free.

```
        /* avatar nitty_bit constants */
        %replace CURSE_IMMUNE by 32
        %replace VOTED_FLAG by 3;
        %replace GOD_FLAG by 4;
        %replace MISC_FLAG1 by 5;
        %replace MISC_FLAG2 by 6;
        %replace MISC_FLAG3 by 7;
```

CURSE_IMMUNE is used to keep one from being infected more than once. VOTED_FLAG is used to prevent people from voting twice in an election. GOD_FLAG is used for superuser avatars. MISC_FLAGs are temporaries.

```
        /* object nitty-bits constants */
        %replace DOOR_AVATAR_RESTRICTED_BIT by 32;
        %replace DOOR_GHOST_RESTRICTED_BIT by 31;
```

Setting these on door objects allows you to prevent avatars or ghosts from going through the door.

Helper Routines

All of the "helper" routines are contained in the 'Misc' directory. Most of them are in the large file 'helpers.pl1'. We will summarize them here.

```
        accessable: procedure(objptr) returns(bit(1));
```

given a pointer to an object, returns 'true' iff the object is accessable to 'avatar', i.e., if it is adjacent to the avatar or in an open container which is adjacent or in a container which is in a container which is adjacent, etc.

```
        announce_object: procedure(objptr);
```

given a pointer to an object, broadcasts a HERE_IS message to everyone in the current region describing the object. Takes care of building the description vector and everything. For use when you create a new object.

```
        at_water: procedure returns(bit(1));
```

Obsolete.

```
        drop_object_in_hand: procedure(whoptr);
```

Takes the object in the hand of the avatar pointed to by 'whoptr' and drops it on the ground at that avatar's (x,y) position in the region. If the avatar is empty handed, this is a no-op. It takes care of sending out messages so everyone in the region knows that this has happened.

```
        auto_teleport: procedure(whoptr, where, entry_mode);
```

Teleports the avatar pointed to by 'whoptr' to region number 'where' using the given entry mode. Works asynchronously, i.e., this is what you do to move an avatar who isn't expecting to be moved. Takes care of informing everyone involved, including the victim and anyone in the region from which he departs. Allowed entry modes are:

```
        %replace WALK_ENTRY by 0;
        %replace TELEPORT_ENTRY by 1;
        %replace DEATH_ENTRY by 2;
```

in general, walking should never be used with 'auto_teleport'. 'DEATH_ENTRY' only applies when the avatar is being killed, which you should never be doing yourself (use 'kill_avatar' instead). In other words, always use 'TELEPORT_ENTRY'.

```
        available: procedure(container_noid, x, y) returns(bit(1));
```

Returns 'true' iff the container slot (x,y) in the container with the given noid is empty, i.e., it is available to have something put in it. Ordinarily, the 'y' value is the only value that matters in terms of indicating container slots. The 'x' parameter only matters with regions (and for regions 'available' always returns 'true'), thus you should usually call 'available' with an 'x' value of 0 and a 'y' value of whatever container slot you are interested in.

```
        cancel_event: procedure(event);
```

Obsolete.

```
        change_containers: procedure(obj_noid, new_container_noid,
                new_position, checkpoint) returns(bit(1));
```

Tries to move the object indicated by 'obj_noid' from its present location to slot 'new_position' in the container indicated by 'new_container_noid'. If 'checkpoint' is 'true' it will checkpoint the object to the database after moving it. It returns 'true' iff it was able to move the object (moving an object out of an opaque container increases the C64 memory usage and so will fail if it would overflow memory).

```
        change_region_fail: procedure(who_noid);
```

This is a procedure that the regionproc calls whenever a region change attempt fails. Its job is to undo various things that are done in preparation for a region change in the expectation that it will succeed. Right now all that really needs to be worried about are the lights, but this routine is a nice hook in case something comes up in the future.

```
        dequeue_player: procedure(whatptr);
```

Obsolete.

```
        destroy_contents: procedure(containerptr);
```

Destroys (i.e., removes from the world) all the objects contained in the container object pointed to by 'containerptr'.

```
        empty_handed: procedure(whoptr) returns(bit(1));
```

Returns 'true' iff the avatar pointed to by 'whoptr' has nothing in its hand.

```
        enqueue_player: procedure(whatptr);
```

Obsolete.

```
        getable: procedure(objptr) returns(bit(1));
```

Returns 'true' iff the the object pointed to by 'objptr' is "getable", i.e., if it would be possible for 'avatar' to pick it up (i.e., it is accessable and it is of a type that it is possible to pick up).

```
ghost_say: procedure(obj_noid, text);
```

Like 'object_say', (see below) but for use when the user is a ghost and so 'avatar' is invalid.

```
goto_new_region: procedure(whoptr, where, direction, transition_type);
```

Makes the avatar pointed to by 'whoptr' go to region number 'where' with the transition type (as explained above under 'auto_teleport') of 'transition_type'. If the avatar is walking, 'direction' indicates which direction he is going. This procedure is called for any region transition. For the asynchronous case it is called by 'auto_teleport'. For the synchronous case it is called by 'avatar_CHANGE_REGION'.

```
grabable: procedure(objptr) returns(bit(1));
```

Returns 'true' iff the object pointed to by 'objptr' may be grabbed from another avatar's hand by 'avatar' (i.e., if it is in the hand of an adjacent avatar and is of a type that is allowed to be grabbed).

```
holding: procedure(objptr) returns(bit(1));
```

Returns 'true' iff 'avatar' is holding the object pointed to by 'objptr'.

```
holding_class: procedure(class_number) returns(bit(1));
```

Returns 'true' iff 'avatar' is holding an object of class 'class_number'.

```
inc_record: procedure(whoptr, record);
```

Increments (by 1) the Hall of Records entry for record 'record' and avatar 'whoptr'.

```
item_value: procedure(itemptr) returns(binary(15));
```

Returns the intrinsic value (e.g., what a pawn machine will pay for it) of the object pointed to by 'itemptr'.

```
kill_avatar: procedure(victimptr);
```

Kills the avatar pointed to by 'victimptr'. Takes care of notifying all interested parties, including the victim. Works asynchronously.

```
lights_off: procedure(whoptr);
```

Turns down the lights in the current region on the assumption that the avatar pointed to by 'whoptr' is leaving (lights go down if he is carrying a lit flashlight out of the region).

```
lights_on: procedure(whoptr);
```

Turns the lights back up.

```
lookfor_string: procedure(sourcestring,substring) returns(binary(15));
```

Like ’index’, but performs a case-independent match.

```
        lowercase: procedure(mixedstring) returns(character(256) varying);
```

Returns a copy of the string ’mixedstring’ with all upper case characters converted to lower case.

```
        max_record: procedure(whoptr, record, value);
```

Sets the Hall of Records record for record ’record’ and avatar ’whoptr’ to the maximum of its current value and ’value’.

```
        object_broadcast: procedure(obj_noid, text);
```

Broadcasts an OBJECT_SPEAK message to everyone in the region to make the object ’obj_noid’ say in a word balloon the string ’text’.

```
        object_say: procedure(obj_noid, text);
```

Sends an OBJECT_SPEAK message to the current avatar (only) to make the object ’obj_noid’ say in a word balloon the string ’text’.

```
        pay_to: procedure(whoptr, amount) returns(bit(1));
```

Try to pay ’amount’ tokens from ’avatar’ to the avatar pointed to by ’whoptr’. Returns ’true’ iff it was able to do this (i.e., if ’avatar’ had sufficient tokens in hand to pay the amount).

```
        random: procedure(top) returns(binary(15));
```

Returns a random number in the range from 1 to ’top’.

```
        random_time_in_the_future: procedure returns(binary(31));
```

Obsolete.

```
        region_entry_daemon: procedure(direction, transition_type,
                old_orientation, from_region);
```

Called by the regionproc on entry to a region. This is the place to put any region entry-dependent actions. ’direction’ is the direction the avatar is walking, if he is walking. ’transition_type’ is the transition type. ’old_orientation’ is the orientation of the region departed from. ’from_region’ is the region number of the region departed from.

```
        schedule_event: procedure(objptr, event_procedure, delay)
                returns(pointer);
```

Obsolete.

```
        set_record: procedure(whoptr, record, value);
```

Sets the Hall of Records record ’record’ for avatar ’whoptr’ to value ’value’.

```
        spend: procedure(amount) returns(binary(15));
```

Tries to spend ’amount’ tokens (out of hand) on behalf of ’avatar’. Returns 0 if unsuccessful, 1 if successful.

```
        spend_check: procedure(amount) returns(bit(1));
```

Returns ’true’ iff ’avatar’ could successfully spend ’amount’ tokens out of hand.

```
        tget: procedure(tokenptr) returns(binary(31));
```

Returns the denomination of the token object pointed to by ’tokenptr’.

```
        tset: procedure(tokenptr, amount);
```

Sets the denomination of the token object pointed to by ’tokenptr’ to ’amount’.

```
        unescape_string: procedure(string);
```

Expands all the Ghu character string escape sequences (’\etc’) in the string ’string’ (used in God-tool magic).

```
        vectorize: procedure(objptr) returns(character(256) varying);
```

Generates a contents vector for the object pointed to by ’objptr’.

```
        wearing: procedure(objptr) returns(bit(1));
```

Returns ’true’ iff ’avatar’ is wearing the head object pointed to by ’objptr’.

More Helper Routines: Sending Messages

To facilitate telecommunications, a variety of messaging routines are defined in ’Misc>messages.pl1’. These send messages from the host to the C64. There are many of these routines, but they fall into four functional groups. Within each functional group, the routines are distinguished only by their parameters (in fact, if PL/1 had a variable argument-count procedure call mechanism, there would only be four routines at all).

```
        n_msg_XXX: procedure(to_objectptr, msg_number, args...);
```

Sends message number ’msg_number’ with arguments ’args...’ to the object pointed to by ’to_objectptr’ on all C64’s in the region EXCEPT the one belonging to ’avatar’. I.e., ’n_msg’ == "neighbor message". If ’to_objectptr’ is ’null()’, the message is sent to the region object.

```
        b_msg_XXX: procedure(to_objectptr, msg_number, args...);
```

Similarly sends a message to all C64’s in the region with no exceptions. I.e., ’b_msg’ == "broadcast message".

```
        p_msg_XXX: procedure(to_objectptr, to_whomptr, msg_number, args...);
```

This one sends the message only to the machine whose user is the avatar pointed to by ’to_whomptr’. If ’to_whomptr’ is ’null()’, it assumes that you mean the current avatar but that the current avatar is a ghost. I.e., ’p_msg’ == "point-to-point message".

```
        r_msg_XXX: procedure(args...);
```

Sends a reply message to the current request to the currently requesting avatar.

In all of the above, 'XXX' determines the format of 'args...'. 'XXX' is either a digit, in which case 'args...' consists of that many integer (bin(31)) arguments, or a digit followed by '_s', in which case 'args...' consists of that many integer arguments followed by a single character string argument. For example,

```
n_msg_0: procedure(to_objectptr, msg_number);
n_msg_1: procedure(to_objectptr, msg_number, arg1);
n_msg_2: procedure(to_objectptr, msg_number, arg1, arg2);
n_msg_2_s: procedure(to_objectptr, msg_number, arg1, arg2, argstr);
n_msg_s: procedure(to_objectptr, msg_number, argstr);
```

etc.

More Helper Routines: Width and Collision Detection

A small number of routines relating to collision detection have been isolated in the file 'Misc>width.pl1'. These are separate because they have a large run-time table that they must refer to. This table describes the graphic characteristics, in terms of size and placement, of all the various objects. It is generated automatically by our C64 disk database generation tools which create the include file 'width.incl.pl1' that is included here.

```
adjacent: procedure(objptr) returns(bit(1));
```

Returns 'true' iff 'avatar' is adjacent to the object pointed to by 'objptr'.

```
check_path: procedure(target_noid, x, y, new_x, new_y, flip_path);
```

Performs a collision detection check on a trajectory from location (x,y) to the object 'object_noid'. The trajectory is a city-block path, i.e., all horizontal movement followed by all vertical movement. 'new_x' and 'new_y' are where we wound up. They will be the same as 'x' and 'y' if there was no collision, or the point of collision if we hit something. It will first try vertical-then-horizontal movement. If it hits something it will then try horizontal-then-vertical movement. If the second try succeeds, it will set 'flip_path' to 'true' to indicate that this happened.

```
elsewhere: procedure(objptr) returns(bit(1));
```

Returns 'true' iff the object pointed to by 'objptr' is neither adjacent nor accessable.

```
here: procedure(objptr) returns(bit(1));
```

Returns 'true' iff the object pointed to by 'objptr' is adjacent, accessable, or in hand.

More Helper Routines: Bit Manipulation

Since PL/1's facilities for performing bit manipulation on integers are dreadful, we have written some routines to do this for us. All of the following operate on bin(15) integers, NOT bit strings. These are defined in 'Misc>bits.pl1'.

```
clear_bit: procedure(num, the_bit);
```

Clears (sets to 0) bit number 'the_bit' in 'num' (bits are numbered with the least significant bit as 0 and the most significant bit as 15).

```
        set_bit: procedure(num, the_bit);
```

Sets (sets to 1) bit number 'the_bit' in 'num'.

```
        test_bit: procedure(num, the_bit) returns(bit(1));
```

Returns 'true' iff bit number 'the_bit' of 'num' is 1.

```
        and_bit: procedure(num1, num2) returns(binary(15));
```

Does a bitwise AND of num1 and num2 and returns it.

```
        or_bit: procedure(num1, num2) returns(binary(15));
```

Does a bitwise OR of num1 and num2 and returns it.

More Helper Routines: Capacity Monitor

A series of procedures maintain a model in the host of the C64's memory capacity utilization. They use the include file 'capacity.incl.pl1' which is generated by our C64 disk database creation utilities. These routines are defined in 'Misc>capacity_monitor.pl1'.

```
        note_object_creation: procedure(class_number, style);
```

Notes that an object of class 'class_number' and style 'style' as been added to the region.

```
        note_object_deletion: procedure(class_number, style);
```

Similarly notes the removal of such an object.

```
        reconstruct_memory_usage: procedure;
```

Rebuilds the memory usage model from scratch.

We have not describes the remaining routines in 'capacity_monitor.pl1' because they are local to that file only.

Generic Actions

Many different classes of objects have, at least in part, the same behavior. For the common cases we have created separate generic behavior procedures. These live in the 'Actions' directory and are grouped into files functionally. Because of a problem with the Stratus debugger, we needed to somehow reduce the number of object files that were linked into the program, so the 'Actions' files are handled in a way that is a little peculiar: There is a file named 'actions.pl1' which simply %include's the various sources. Since the brain-damaged Stratus PL/1 compiler insists that include files must have names ending in '.incl.pl1', there are a bunch of '.incl.pl1' files in 'Actions' directory which are simply links to the corresponding '.pl1' files. Here is what is here:

actions_clothing.pl1:
        generic_WEAR
        generic_REMOVE

Behaviors for putting on and taking off "clothing". The reference to "clothing" is historical. All that is left to put on or take off are heads, which use these routines.

actions_container.pl1:
        generic_CLOSECONTAINER
        generic_OPENCONTAINER
Behaviors for opening and closing containers. These worry about locks and keys too.

actions_door.pl1:
        generic_CLOSE
        generic_OPEN
Similarly, behaviors for opening and closing doors.

actions_gpt.pl1:
        generic_GET
        generic_PUT
        generic_THROW
The most common behaviors of all, for picking up, putting down, and throwing objects.

actions_help.pl1:
        generic_HELP
Behavior for getting help. Objects which only require a fixed-string help message use this. The bulk of this procedure is a giant text array with one entry for each possible class. I’m sure this wastes a lot of memory, but PL/1 doesn’t allow us to declare static text arrays properly. More on help below.

actions_mail.pl1:
        generic_READMAIL
        generic_SENDMAIL
Behavior for mail. Obsolete, I think.

actions_oracle.pl1:
        generic_ASK
Behavior for talking to oracular things.

actions_switch.pl1:
        generic_OFF
        generic_ON
Behavior for turning switchable objects on or off.

actions_weapon.pl1:
        generic_ATTACK
Behavior for using weapons.

Major Subsystems

There are three major subsystems underneath the behavior code which are complex enough to deserve special discussion on their own. These are the mechanisms for magic, sensors and drugs; the help system; and the "curse" system.

Magic, Sensors and Drugs

Magic objects, sensors, and drugs all employ essentially the same mechanism. These classes are distinguished by the fact that a given type of object can have one of a number of possible (very different) behaviors. In each case, the object record contains a type number that indicates just exactly what sort of magic item, sensor or drug the object is. This number is used as an index into an array of procedure entry points which fans out to a number of procedures which then implement the specified function. Beyond this, there are some slight differences between the three types of variable-behavior object:

Sensor routines live in 'Classes>class_sensor.pl1'. The dispatch array is initialized by the procedure 'initialize_sensors' which must be called by the regionproc at system start-up time. The sensor routines are called by the class sensor behavior routine 'sensor_SCAN'. Such routines should look about the region for some characteristic of interest and then return a 1 or 0 depending on whether or not they find it. This success/failure value is then transmitted back to the C64 by 'sensor_SCAN'.

Drug routines live in 'Classes>class_drugs.pl1'. The dispatch array is initialized by the procedure 'initialize_drugs' which must be called by the regionproc at system start-up time. The drug routines are called by the class drugs behavior routine 'drugs_TAKE'. By convention, such routines should take some action effecting the player's avatar 'avatar'. In general, they should not effect anyone else's avatar or the region environment. Drug routines do not have to worry about sending a response message to the player, as this will be taken care of by 'drugs_TAKE', though they DO have to worry about sending any asynchronous notification messages regarding any specific actions they perform. 'drugs_TAKE' also worries about whether there are any pills left in the pill bottle when the user tries to take one, and about decrementing the pill count after one is taken.

Magic routines live in 'Misc>magic.pl1'. The dispatch array is initialized by the procedure 'initialize_magic' which must be called by the regionproc at system start-up time. The magic routines are called by the generic behavior routine 'generic_MAGIC' which, violating the above described conventions about the 'Actions' directory, is also located in 'Misc>magic.pl1'. 'generic_MAGIC' sends out an unconditional success response and dispatches to the appropriate magic routine. Thus, magic routines are running free of the C64 which has already recieved a response to its request. This is significant if the user issues some other request in the meantime (such as leaving the region).

There are actually two types of magic objects, "switches" and portable magic items. In the case of the latter, a parameter is supplied by the C64 when requesting magic action that is the noid of the object or avatar at which the player is pointing with his cursor when he issued the request. This allows the action of magic objects to be directed at or against something specific. The helper routine

```
avatar_target_check: procedure(targetptr) returns(bit(1));
```

is defined in 'Misc>magic.pl1' to check if the thing pointed at by the user is another avatar, since frequently one wishes to have magic which operates on avatars only. The range of possible actions that a magic routine may take is almost unlimited; the interested reader is advised to look at the source file 'Misc>magic.pl1' itself for examples of the sorts of things we can and do do with magic.

Curses

A less important but still significant sub-system is the curse mechanism. This is what we use to implement "cooties" as well as other sorts of plagues. A curse temporarily modifies the attributes of an avatar. Each avatar record has two fields for dealing with curses, 'curse_type' and 'curse_count'.

Normally, 'curse_type' is 0, meaning no curse. However, if 'curse_type' is not 0, the avatar has some temporary attribute that he is (probably) trying to get rid of. An avatar becomes cursed by a mechanism that varies with the type of curse. Typically it is started by some sort of magic. The routine that starts the curse must set the curse fields appropriately. To help, the file 'Misc>curses.pl1' defines the procedure

```
activate_head_curse: procedure(victimptr, curse_type) returns(bit(1));
```

This procedure attempts to inflict the curse 'curse_type' on the avatar 'victimptr'. It returns 'true' iff it succeeds. This procedure specifically deals with curses whose manifestation is a weird head of some sort. It takes care of notifying the player that he has a new head.

Curses are typically contagious. The transmission of curses is handled by the routine

```
curse_touch: procedure(curserptr, curseeptr);
```

which is called by 'avatar_TOUCH' when one player tags another while cursed. This routine transmits the curse (if it is contagious) from the avatar 'curserptr' to the avatar 'curseeptr' and decrements (if appropriate) 'curserptr's 'curse_count' field. When this counter runs down to zero the curse is removed and the avatar's old head is restored. By the way, when an avatar loses a curse a bit is set in the avatar record that makes him immune to getting it again until we reset the game. Of course, the curse is only transmitted by a tag if the victim is not himself immune by this means.

By controlling the initial setting of 'curse_count' when an avatar is given a curse, you can manipulate the nature of the spread. Setting it to one gives a curse that passes from player to player, such as cooties. Setting it to a small number (such as two) causes a plague that spreads exponentially. Setting it to a large number causes it to infect the whole population eventually.

Both the above mentioned routines in 'Misc>curses.pl1' have case statements in them that vector on the curse type, so that curse specific actions may be taken.

Help

The help system is invoked when the player presses the F7 key. The basic action to be taken by any help behavior is simply to send out a response message with a character string of up to 114 characters in length. However, owing to the variety of things we might want to say about an object in its help message, there are a number of complications.

If the help message associated with a particular class of object can be expressed in a string of up to 114 characters whose text never changes, then you should set the class's HELP behavior to 'generic_HELP'. 'generic_HELP' contains an array of strings, which it indexes by class number. When called, it looks up the object's class and transmits the appropriate string. There are a few special purpose entries in this array, however. If a class does not exist its help array entry should be '-'. If a class exists but does not use the 'generic_HELP' routine, its help array entry should be 'i'. In both cases it triggers an error diagnostic, since these help messages should never be encountered. If you haven't gotten around to figuring out what a class's help message should be, set its entry to 'u'. This will give an appropriate apology for there not being any help. Finally, if an object is a non-functional scenic object, set its help entry to 's'. This will give help that describes how to use HELP.

If an class's help information cannot be expressed in 114 characters or if it must vary depending on other state information, then the class needs its own help behavior. Long help messages can be accomplished simply by breaking the help text into multiple messages. The first is a response message that answers the HELP request itself, while the remaining messages should be sent with calls to 'object_say'. Variable content messages can simply be built up as needed and sent via the same means.

Certain types of objects have HELP information requirements which are more complex still. In particular, classes which exhibit large stylistic or functional variations require special treatment. Such classes include drugs, sensors, knick-knacks, and all magic items. These classes have arrays of messages of their own which are indexed by style, magic type, or whatever other parameter is appropriate for the class in question. When adding a new type of magic or a new sensor, then, you must also add an entry to the appropriate help text array. By the way, it is our convention that the help for sensors and drugs should be descriptive while the help for a magic item should be phrased as a riddle or cryptic remark that only hints at what the magic item does.

The final complexity is introduced by vending machines. Vending machines issue help message which describe not only how to use the vending machine but what it is that is for sale. As with 'generic_HELP', 'vendo_HELP' (located in 'Classes>class_vendo_front.pl1') maintains an array of help messages. This array is index by the class of the object on display in the vending machine. It works pretty much just like 'generic_HELP' except that the messages are limited to 80 characters. Also, like 'generic_HELP', there are some special entries in the array which cause special action to be taken. Not only do we have to deal with non-existent classes and such, but the problem of the variability of knick-knacks, magic items, and so forth creeps up on us once again. To handle these cases, a number of procedures with names of the form 'classname_vendo_info' are defined which return appropriate character strings based on a pointer to the object on display. 'vendo_HELP' calls these based on the entries in the vendo help message array. Here are the special entries in this array:

'-' means that the object class does not exist. Hitting such an entry is a run-time system error.
'i' means that the object is a class that may not be placed in a vending machine. Hitting such an entry is a run-time system error.
'b' uses whatever is returned by 'book_vendo_info'
'd' uses whatever is returned by 'drugs_vendo_info'
'm' uses whatever is returned by 'magic_vendo_info'
'k' uses whatever is returned by 'key_vendo_info'
's' uses whatever is returned by 'sensor_vendo_info'

In the case of drugs, magic and sensors, the information used by the 'xxx_vendo_info' routine and the information used by the 'xxx_HELP' routine comes from the same array which does double-duty.