Habitat Object Disk Version Management and Field Update

The C64 Habitat software is designed so that the object disk can be updated in the field automatically by the host. The basic mechanism is a track/sector block write message that allows selected blocks of the object disk to be replaced by new data. However, to use this update technique efficiently will require some support software on the host to manage version update. Here is a general overview of the proposed mechanism:

The host will keep a database that contains the contents of the Habitat object disk image. There will be one record for each disk block. Each disk-block record will contain, in addition to the 256 bytes of data in the block, the track/sector number of the block and a version ID number for the block. Whenever a new version of the object disk is released, those blocks (and only those blocks) whose contents have changed will be written into the disk-block database along with the new version number. Thus, each record in the database is tagged with the version number of the oldest version of the object disk for which that block is up to date.

Whenever a user enters Habitat, his C64 announces his presence with a message that contains, among other things, the version ID of the object disk that he is using. The host, before allowing the user to enter the system, sends a series of block-update messages, one for each block in the database that has a version number newer than the version that the user logged on with. The last block-update transmitted should the one that modifies the block on the object disk which contains the version number itself, so that if the update sequence is interrupted the process can be restarted from the beginning with another login.

Here are specific details:

When the user enters Habitat, he announces his presence with the IM_ALIVE message:

```
        <REGION><IM_ALIVE><program disk #><data disk #><data disk version #>
```

<REGION> is 0, i.e., the message is addressed to the region object on the host. <IM_ALIVE> is the message number, 3. The three parameters are the ASCII ID’s for the disk that is trying to enter. The current program disk number is "2"; the current data disk number is "0" (note that these are ASCII values, i.e., the program disk number byte is 0x32). These numbers won’t vary unless we issue an entirely new product based on the Habitat software. The data disk version number is the number of interest. Right now it is "0", but will change as new versions are released.

The host version database records should look something like:

```
declare 1 disk_block_record based,
                2       track           character(1),
                2       sector          character(1),
                2       version         character(1),
                2       data            character(256);
```

where ’track’ and ’sector’ together are the primary key. ’version’ may also want to be a key so that you can selectively read only those blocks that need to be sent out in an update.

The update message that the host sends out has this form:

```
        <REGION><UPDATE><track><sector><data...>
```

where <REGION> is 0, i.e., the message is addressed to the region object on the C64; <UPDATE>, the message number, is 11; <track> and <sector> are the track and sector to update; and <data...> is 256 bytes (exactly) of data to be written to the specified disk block.

It doesn’t matter what order the disk blocks to be updated are sent to the C64, as long as the block containing the version ID is sent last. This is track 0x12, sector 0x0. The version ID block has a very specific format. You probably won’t need to worry about what this is since it will be in the database, but we’ll describe it anyway, just in case. It is an ASCII string exactly 256 characters long, that has the form:

```
"HI HACKER! DISK 0 VERSION 0 HABITAT (c) ... <credits here out to 256 bytes>"
```

The important points are that the "HI" is an ID string that the C64 program looks for on boot, and the disk and version numbers must be located at precisely the offsets in the string that they have here. The remainder is arbitrary, but of course we like to have a copyright notice in there.

Here is some PL/1-oid pseudo-code for the update procedure:

```
procedure field_update(version);
     declare version character(1);
     declare 1 a_block like disk_block_record;
     %replace ID_TRACK by 18; /* 0x12 */
     %replace ID_SECTOR by 0;
     declare ID_TRACK_SECTOR character(2) init(ID_TRACK, ID_SECTOR);

     if (rank(version) < Current_version_number) then do;
          position_block_database(version, ’version’, KEY_GT);
          do while (read_block_database(a_block) ^= end_of_file);
               if (a_block.track^=ID_TRACK & a_block.sector^=ID_SECTOR) then
               send_block_update(a_block.track, a_block.sector, a_block.data);
          end;
          position_block_database(ID_TRACK_SECTOR, ’track_sector’, KEY_EQ);
          read_block_database(a_block);
          send_block_update(ID_TRACK, ID_SECTOR, a_block.data);
     end;
end;
```

This would be called with the version byte extracted from the IM_ALIVE message. ’Current_version_number’ is a global that maintains the most up-to-date version number of the object disk. Actually, you would probably want to perform the check against this value BEFORE you call this routine, so as to save a transaction with the database process in the typical case. The ’position_block_database’ call positions the block database to read the records as specified. ’read_block_database’ simply reads a block database record into the given variable and returns an indicator if it has read the last record. ’send_block_update’ sends an update message in the form specified above.

At Lucasfilm we have utilities to layout new versions of the object disk. When there is an update to be distributed, we will provide a new disk image in the form that we have been using in our current telecommunications release procedure, i.e., a set of ".dat" files each of which has a predefined track/sector offset location on the image disk. It will be necessary to do a block-by-block comparison of the new disk data with that in the block database, and update (only) those records which have changed by writing the new data and the new version number.

Our utilities are designed so we create a new object disk image with the smallest possible number of changes from the old one, so that the number of update messages that must be sent is minimized.