is the `photo` attribute. Photographs of people and objects are stored in the directory as a g3fax-encoded block of ASN.1, and are best stored as "file" attributes described in the next section. There is an example of a photograph attribute in the example database. This can be looked at by connecting to the directory with DISH and looking at the entry "Steve Kille, CS, UCL, GB".

Unless you you have compiled and installed the "photo" code as described in Section 12 you will see the message

```
"No display process defined"
```

Having compiled the "photo" code, you will need to add a line such as

```
photo xterm Xphoto
```

to your *dsaptailor* file (see Section 13.2) This says that if the user in logged on to a terminal of the type `xterm` then use the process *g3fax/Xphoto* in the ISODE `ETCDIR` directory to display the photograph.

There are display routines provided for the X Window System, SunView, Tektronix 4014, and "dumb" terminals.

Generating Photos

The photograph files used by QUIPU need to be two-dimensionally g3fax-encoded. QUIPU contains some tools to help you get your files into the required format. These are found in the *others/quipu/photo* directory of the ISODE source tree.

Getting pictures onto your machine is a local problem that will need discussing with your system manager. You may have them in a different format from that required by QUIPU.

If they are in SunView *pixrect* format, the tool `pr2pe` can be used to convert them. For other formats, there are two utilities, `pbmtofax` and `faxtopbm`, which can convert from the "PBM" format to two-dimension fax, and vice versa. The PBM format is that defined and used by Jef Poskanzer's "pbmplus package". This package can convert a large range of bitmap formats into the PBM format[6]. You will need to obtain this package to compile the `pbmtofax` and `faxtopbm` utilities.

[6]You should not confuse `pbmtofax` with the `pbmtog3` tools which are part of the pbmplus package; this uses a one-dimensional encoding scheme

Note: with this version of QUIPU, the `pbmtofax` tool should always be used with a `-old` flag to ensure user at remote sites with older code can still decode and display your photograph files.

14.7.4 File Attributes

Attributes values are generally stored in the EDB file, and loaded into memory when the DSA starts. For some large attributes such a photos, this is not a sensible approach, so the concept of a "file" attribute is introduced.

If an attribute value is prefixed by `FILE`, then the value is assumed to be stored on disk. For example, if the following were found in an EDB file:

```
photo={FILE}/usr/local/pictures/steve
```

then the value for the photo attribute would be read from the file

`/usr/local/pictures/steve`.

The syntax of the data in the file is expected to be the same as the string syntax, except in the case of ASN.1 and photo attributes which are expected to be in raw ASN.1.

If there is not a file name supplied, then a default name is allocated. The default file name is the RDN of the entry the attribute belongs to followed by a dot ".", followed by the attributeType. This file is expected to be in the same directory as the EDB file for the entry. For example the default file for photo attribute, representing the entry for "c= GB @ o= University College London @ ou= Computer Science @ cn= Colin Robbins" would be:

```
cn=Colin Robbins.photo
```

in the directory

```
quipu-db/c=GB/o=University College London/ou=Computer Science
```

or equivalent mapped file as described in Section 14.1.4.

The process defined so far allows for attributes stored in an EDB in file format to be read by a DSA. Writing the attributes back to files if modified/added by a DUA requires a little more. The DSA needs to know which attributes should be stored on disk. This information is supplied in the attribute "oidtables" which are defined in Section 14.11. For example if the following entry was found in the file *oidtable.at* in the ISODE `ETCDIR` directory then a `photo` attribute would always be written back to disk.

```
photo:  thornAttribute.7: photo : file
```

14.7.5 Attribute Inheritance

Attribute inheritance is a mechanism whereby an entry can get default attribute values from its parent entry. This can make management of common attributes much easier. Inheritance can be used to make the in-core database significantly smaller, which as a side effect should make the DSA run faster. For example, entries of the object class `person` for a particular organisation might all have the same postal address attribute. Using inheritance this attribute can be placed in the organisation entry and inherited down to all `person` entries.

```
InheritedListSyntax ::= SET OF CHOICE {
   AttributeType,          -- Take value from the entry
   Attribute }

InheritedAttributeSyntax ::= SET {
   default [0] InheritedListSyntax OPTIONAL,
      -- default which can be overriden in lower entry.
   always  [1] InheritedListSyntax OPTIONAL,
      -- always present in lower entry.
   objectclass OBJECT IDENTIFIER OPTIONAL                    10
      -- object class to inherit to.
      -- The null case means the objectclass attribute
      --    itself is inherited
}

InheritedAttribute ATTRIBUTE
   WITH ATTRIBUTE-SYNTAX InheritedAttributeSyntax
   MULTI VALUE
```

Figure 14.4: Attribute Inheritance

The information is placed in the entry above using the `inheritedAttribute` attribute type, defined in Figure 14.4. The QUIPU string syntax to represent this is:

```
<InheritedAttributeValue> ::= [ <oid> "$" ]
                        [ "always" <InheritedList> ]
                        [ "default" <InheritedList> ]
<InheritedList> ::= "(" NEWLINE <AttributeSequence>
                          NEWLINE ")"
```

For example, if the following was an attribute of an entry

```
inheritedAttribute = person $ always (
postalAddress= UCL $ Gower Street $ London $ WC1E 6BT
postalCode= WC1E 6BT
) default (
telephoneNumber= +44 71-387-7050
)
```

then all entries of objectclass `person` positioned one level below the entry would always inherit the attributes

```
postalAddress= UCL $ Gower Street $ London $ WC1E 6BT
postalCode= WC1E 6BT
```

and **if** the entry does not contain a `telephoneNumber` attribute itself, it would inherit

```
telephoneNumber= +44 71-387-7050
```

The attribute value in the inherited attribute can be left blank, for example

```
inheritedAttribute = person $ always (
postalAddress
)
```

in which case the value is taken from the same entry as the inherited attribute itself. This avoids duplicating attributes.

The OID representing the object class can only appear in one attribute value, thus the following, which may seem similar to the above is *not* allowed: the attributes need to be combined.

```
inheritedAttribute = person $ always (
postalAddress= UCL $ Gower Street $ London $ WC1E 6BT
)
inheritedAttribute = person $ always (
postalCode= WC1E 6BT
)
inheritedAttribute = person $ default (
telephoneNumber= +44 71-387-7050
)
```

If the OID representing the object class is omitted, then the values are inherited to entries that themselves do not contain an object class attribute. Thus the inherited attribute must define an object class to inherit down (as schema checking still takes place). For example

```
inheritedAttribute = default (
ObjectClass= person & quipuobject
postalAddress= UCL $ Gower Street $ London $ WC1E 6BT
postalCode= WC1E 6BT
)
```

Inheritance can be used for any attribute with the following exceptions:

- An entry cannot inherit its distinguished attributes.
- An inherited `userPassword` attribute in not used to authenticate a user at bind time.
- The following system attributes

objectClass	ACL
masterDSA	slaveDSA
aliasedObjectName	edbInfo
presentationAddress	relayDSA
treeStructure	supportedApplicationContext
inheritedAttribute	

  can only be used in the "default" clause. They can not appear in the "always" clause as there is the potential for an inconsistency to occur.

- You must hold at least one copy of the EDB file containing the parent entry.

If an entry belongs to more than one object class, and more than one of these classes is listed in the inherited attribute of the parent entry, then only attributes from *one* of these will be inherited into the entry. It is undefined which.

The inheritance described thus far only covers one level of the DIT, whereas it is often useful to inherit attributes down the whole subtree. This can be achieved by inheriting the inheritance attribute! Consider the multi-valued attribute

```
iattr= organizationalUnit $ default (
iattr
)
iattr= person $ default (
telephoneNumber
)
```

Every `person` immediately below will inherit the telephone number attribute. Every `organizationalUnit` will inherit the same inheritance attribute, thus any `person` entries below `organizationalUnit` entries will get the telephone number attribute as well. by the same token, this works for multiple levels of `organizationalUnit`.

14.8 How a DSA Starts

The section gives a brief description of how a QUIPU DSA starts. This is intended to help people who have experienced problems in getting their DSAs started.

First of all the tailor file *quiputailor* in the ISODE `ETCDIR` directory is read (unless you specify a different tailor file using the `-t` option to *ros.quipu*). This tells the DSA, amongst other things, its own name, and its parent DSA(s).

The first thing a DSA need to do is find its own entry, thus it tries to load the EDB that should contain it. If it cannot find its own entry, then it will try connecting to the parent DSA to read its own entry. If this fails the DSA will stop.

Having found its own entry, the DSA will check to see if it is a cached entry read fron disk; if so, it will attempt to read a new version from the relevant remote DSA. If successfu,l a new cache EDB file will be written.

Now the rest of the local DIT can be loaded. Each EDB specified in the `edbInfo` attribute is in turn loaded from disk, followed by any cached EDB files.

As each EDB is loaded from disk, it is checked to make sure all the attributes in the entry are allowed, as defined by the `objectClass` attribute, and that the tree shape conforms to that defined by the `treeStruture` attribute. If any EDB fails these checks then the DSA will not start, and an appropriate message will be logged.

Having loaded all its data, the DSA will start listening for DAP and DSP associations.

14.9 Adding More DSAs

Many organisations will need to have more than one DSA to meet their requirements. This section suggests how you might arrange your DSAs to get the most out of the system.

You will almost certainly need at least one DSA that holds a copy of the root EDB, and your country EDB. This DSA should also hold the master copy of your organisation's own EDB. Subsequent DSAs that are set up should have this DSA defined as one of their parent DSAs.

For robustness, it is a good idea to have some other DSAs replicating these EDBs, so that if one local machine or DSA becomes unavailable, then there is another copy elsewhere that can be used. EDBs that contain addresses of other DSAs that you may want to contact regularly should also be replicated, to prevent extra associations having to be made just to read DSA addresses. In short, if you know that data from a certain EDB is going to be accessed a lot, replicate it locally. Replication does not have a high cost.

When setting up multiple DSAs be sure to name them as described in Section 14.2.2; ensure the DSA entries are sufficiently high in the DIT, so that other DSAs can read the entry without having to contact the DSA concerned.

14.10 Receiving EDB Updates

If your DSA holds a slave copy of one or more EDBs, then it can automatically update these for you. The current approach is very simple minded, but will be extended for future versions.

There are two ways to ask a DSA to update its slave EDBs, either use DISH program and issue a `dsacontrol -slave` command (see Section 4.5), or set the "quiputailor" parameter "update" to **on** (see Section 14.3), in which case the DSA will try to update its slaves when it starts up.

To update the slaves, the DSA uses the `edbinfo` attribute that the DSAs entry in the DIT *must* have. This attribute specifies which EDBs you hold, and where to get updates of the from. *Note*: you do not necessarily have to get updates from a master EDB, a slave is acceptable. In many cases this will be preferable to prevent the load on the master DSA being too high. For example, a "national" EDB is likely to be highly replicated, it would not be a good idea to have just one DSA handling updates. It would be better to have the load spread over several DSAs.

The syntax of the attribute is

```
edbinfo = EDB concerned # get from # send to
```

It is a multivalued attribute.

A few examples:

```
# cn=Giant Tortoise # cn=Fruit Bat
c=US # # cn=Fruit Bat
c=US # # c=US@cn=Spectacled Bear
c=US@l=NY # cn=Fruit Bat #
```

Note that there is no harm to using multiple `eDBinfo` lines, even if they refer to the same EDB. These lines indicate that:

- The root EDB is read from the `cn=Giant Tortoise` DSA, further the `cn=Fruit Bat` DSA is allowed to read the root EDB from this DSA. This is an important point — A DSA has to explicitly say it will allow you to update from it, before it will send EDB files. Thus if you want to pull EDB files from a remote DSA, you will need to ask the manager to add your DSA to their DSAs "send to" field.

- The `cn=Fruit Bat` DSA and the `c=US@cn=Spectacled Bear` DSA are allowed to read the EDB for `c=US`. Note that the second and third line could be combined as:

```
c=US # # cn=Fruit Bat$c=US@cn=Spectacled Bear
```

- The DSA-named `cn=Fruit Bat` supplies the EDB for `c=US@l=NY` to this DSA.

Some EDB files, such as the ROOT, and country level EDB files are highly replicated, and having to list all the DSAs that are allowed to pull such EDBs is too restrictive. Thus the following syntax can be used as an abbreviation:

```
c=US # # c=US
```

This indicates that the c=US EDB file can be sent to DSA named at the c=US level of the DIT, for example;

```
c=US@cn=Wombat
```

is allowed to pull the EDB, but the following are not:

```
cn=Wombat
c=US@o=Foobar Inc.@cn=Wombat
c=GB@cn=Wombat
```

Whether to update is decided upon by looking at the version string; if the two EDB files have a different version string, then the EDB will be updated.

14.11 Tables

Throughout QUIPU, all the Object Identifiers that need to be specified are specified using a string representation, for example:

String representation	Object Identifier
attributeType	2.5.4
surname	2.5.4.4
streetAddress	2.5.4.9
organizationName	attributeType.10

A set of Object Identifier tables in the ISODE `ETCDIR` directory are used to provide this mapping (see also Section 9 in *Volume One*). In general the default table will be sufficient. This section describes the tables for those who want to extend the default definitions.

This basic format is used to build up the specification of general object identifiers. This file is by default named *oidtable.gen*, and has simple mappings from string to oid. These strings can then be used in the definition of further oids (for example the `organizationName` oid in the table above).

This simple table is extended to give table formats for attributes This file is by default named *oidtable.at*. Each entry in this table is assumed to represent an attribute, so in addition to mapping strings onto oids, it also defines the syntax for that attribute:

String representation	Object Identifier	Syntax
objectClass:	attributeType.0:	objectclass
aliasedObjectName:	attributeType.1:	dn
commonName,cn:	attributeType.3:	caseignorestring
searchGuide:	attributeType.14:	asn
x121Address:	attributeType.24:	numericstring
presentationAddress:	attributeType.29:	presentationaddress

This says that the attribute named `objectclass` represents the oid "attributeType.0"; using the expansion of "attributeType" (as defined in the file *oidtable.gen*) thus gives the oid "2.5.4.0". The syntax taken by the attribute value is `objectclass`.

Similarly an `aliasedObjectName` has `dn` (DistinguishedName) syntax. The recognised syntaxes are defined in the BNF in Appendix B.

After the "syntax", an extra optional parameter is allowed; if this is the string "`file`" the the relevant attribute is designated a file attribute; see Section 14.7.4 for a description of file attributes.

The file *oidtable.oc* defines the object classes QUIPU knows about. The file again maps strings onto oids. Each string is assumed to represent an object class, and for each it defines the hierarchy (which object classes it is a subclass of), the mandatory attributes of the object class, and the optional attributes of the object class.

For example the entry

```
quipuObject:   quipuObjectClass.2 : top : acl : \
```