KinDB Reference

Table of Contents

Kin Record Definition

Note: Here we talk about records, as a group of fields. Record is roughly equivalent to standard data base table definition, with one record values spread in every table row, and field would be equivalent to data base column for a single row.

Predefined fields

There's some fields that are defined always, even when they are not present at the record definition. This is because system needs them for internal use (thus, if you define them, they will be auto assigned and read only). If you don't define them, you still may read them although not explicitly defined (won't appear if you request that record definition). Currently, there's only one:

ID

Always part of a record, this is a unique identifier (for that "row"), so it can be referenced by any other field of this type into another record. This includes standard records, and also variable length items (a text string, for an instance). If not specified, this is ***rID*** type (32bit). If you know you don't need so many records, you may define it as ***sID*** (16bit). There's automatic conversion from sID to rID, but the reverse may overflow (thus reporting reference error).

Predefined subfields

There's some types or attributes that implicitly define some additional data for a field. Access to such data may be performed by the means of a few predefined (and thus reserved) sub-fields. These are:

who

for a historical field, this is defined as the user ID who modified this field. You can't specify this as an array (order does not matter here), but any value that matches this. You shouldn't define a field called like this, to avoid masking it when parent field is historical.

when

same as previous, but this is the timestamp when a change was made.

Predefined types

Available field types (and their corresponding type):

Numeric types

Integer types

Signed

- 8bits (***sByte***)
- 16bits (***sWord***)
- 32bits (***sInt***)
- 64bits (***sLong***)

Unsigned

- bitmap (*1 to 64 bits* ***BitMap***)
- 8bits (***Byte***)
- 16bits (***Word***)
- 32bits (***Int***)
- 64bits (***Long***)

Unsigned sets

- 8bits set (***bSet***)
- 16bits set (***wSet***)
- 32bits ser (***iSet***)

IDs

- 16bits (***sID***, ***rsID***)
- 32bits (***rID***, ***rrID***)

Version control

These fields are automatically incremented every time the record is modified. Note that if no field changes, no increment is performed.

- 16bits (***msSN***)
- 32bits (***mSN***)

Accounting

These types are useful for usage counters. They perform an addition of supplied value to the stored one, instead of a replacement. Usually, supplied value is '1', but bigger or even negative values may be supplied. These fields are unsigned integers, and operations are saturated (e.g. isCT=65535, any positive number added will not wrap and will stay at the max. value).

- 16bits (isCT)
- 32bits (iCT)

Floating point

- Half precision (16bit IEEE 754r) (***fp16***) (6.10E-5 to 65504), not supported yet (and may last unsupported on some architectures for a long time).
- Single precision (32bit IEEE 754) (***fp32***) (1.175494351×10−38 to 3.4028234 x 1038)
- Double precision (64bit IEEE 754) (***fp64***) (2.2250738585072020×10−308 to 1.7976931348623157×10308)

Fixed point

Money and sMoney types rely on a table of currency definitions, with the international currency code (3chr), utf-8 character representation, # of decimals (EXP), and current change rate to the reference currency (32bit single precision).

Currency values are always expressed as COD12345.67 (COD is currency standard code, like USD or EUR, and the value has no thousands separators, and a dot as decimal delimiter).

- Currency: 40bits signed value, plus 8bits currency code (***sMoney***). Fixed point for only the decimal digits specified for given currency code (EXP).
- Currency: 56bits signed value, plus 8bits currency code (***Money***). Fixed point for the decimal digits specified for given currency code (EXP).

Date types

Standard

Timestamps and date/time values with different resolutions and valid ranges. For time values, they may represent both time of the day, or duration (if not too long).

- Unix time stamp (***uDateTime***): seconds since 1/1/1970 (32b) (ranges up to 5/2/2106 approx.). This is being revised, to adhere to strict 32bit unix style (signed, and thus going from 1902 to 2038 aprox.)
- Extended time stamp (***xDateTime***): minutes since 1/1/5000bc (32b) (ranges up to 15/12/3165 approx.)
- Short Date (***sDate***): days since 1/1/1900 (16b) (ranges up to 4/5/2079 approx.)
- Time (***Time***): 1/10000 seconds since 0:00:00.0000
- Short Time (***sTime***): double seconds since 0:00:00 (i.e. even seconds)

Values are often expressed with a prefix to indicate its type:

- *d*: like in d20040815, for a date (August 15th, 2004). Applicable on an sDate, uDateTime, or xDateTime types
- *t*: like in t193559 (7:35:59pm) or in t193559.3809 (same but with 380.9 milliseconds

more), for a time. First case applicable on Time or sTime types, the former only for Time type

- *x*: like in x2903092851, for an extended time absolute value -(inutes since 1/1/5000bC). Only applicable to xDateTime type
- *u*: like in u1273055960 (May 5th, 2001, 10:39:20 UTC). Applicable to uDateTime type.
- No preffix: like in 20040815180959, for a complete (August 15th, 2004, 6:08:59pm) Date+Time value. Applicable on an uDateTime, or an xDateTime.

Automatic

Timestamps assigned by the system (read only by the user):

- Creation date (***uDTcrea***): like uDateTime, but read only and assigned automatically on record creation.
- Modification date (***uDTmodi***): like uDateTime, but read only and assigned automatically on every record modification. Note that if no field changes, this date is not updated.

GIS types

These are geographic positioning types:

- 2D (***G2D***): coordinates (latitude, longitude), in 1/3300 of second (approx. 1cm resolution at Ecuador). Expressed like in (N40D2M50S9849,E1D26M15S0712) (N=North, E=East, D=degrees, M=minutes, S=seconds)
- 3D (***G3D***): coordinates (latitude, longitude) like G2D, plus height over mean sea level (in 50cm steps, so from 16km under water to 16km over it). Expressed like in (N40D2M50S9849,E1D26M15S0712,615m5) (40º 2' 50.9849” North, 1º 26' 15.0712” East, at 615.5 meters over sea level)
- 2D diffuse (***G2Dd***): like G2D, but with an influence radius or area “size”. Expressed like (N40D2M50S9849, E1D26M15S0712, R500m5)
- 3D diffuse (***G3Dd***): like G3D, but with an influence radius or volume “size”. Expressed like (N40D2M50S9849, E1D26M15S0712, 615m5, R500m5)

User stamp types

There are two automatic fields that refer to the *_User* record (table). These are:

- Modification User (***muID***): automatic read only field, updated with the UserID that modifies a record that declares this field. Note that it is only updated if at least one field changes its value.
- Creation User (***cuID***): automatic read only field, set with the UserID that creates a record that declares this field.

See Appendix 2 for required records and fields.

Binary data types

These reference types point to a binary, variable size block object that holds arbitrary (but partially classified) content:

- Generic Static Image (***Image***).
- Generic Block (***Block***[MaxLength]): up to MaxLength bytes of binary storage (referenced object). The object definition also declares its granularity (or item size), often *Base8* (8bit data), but other sizes are allowed (16 and 32bit).

Structure declaration

1. Declaration arrangement tricks:
   - Arrays ([])

     A way to define groups of several sequential identical fields.
   - SubRecords (SubRecord)

     A way to arrange data into a definition (to encapsulate it). Actually, data defined into the clause is exactly the same as if it was defined outside it. The difference is that you may use the same name for two fields in one record if they belong to different subrecords (scopes), and field name includes one or more dots.
2. Objects Declaration:
   - Record (Record)

     Defines an object composed by fixed size embedded fields, and variable length referenced fields (that implicitly require their respective objects to exist).
   - Text Objects (Object)

     Declares a text management object (to be loaded by the system). A text subtype must be specified (see Text Types), and a module should exist with methods to manage that subtype.
   - Binary blocks Object (Object)

     Declares a variable length binary (user data) block object. Only one subtype is defined (block), with its corresponding manager.
   - Image Object (Object)

     Declares an image object (graphical image, like a map, photo, drawing...). Only a generic subtype is defined by now (image), but it will probably include several attributes.

Stateful data

Some kinds of data types might be declared as state sensitive, that is, they will retain all the values they're assigned to, along with information about when the values changed and who did it (user ID). This is similar to a set (which may hold multiple values for a field), but it also includes (automatically) information about when and who, and it allows to get the value of

the field into any fixed moment in -past- time (specified on the query) thus behaving like it had a single value, but value returned may differ depending on the time spot we request. Typical examples are state of something: it may change with time, and we can request state (along with other fields) of that stuff in some past time, so we get state at that moment, not current state. This allows to calculate statistics for any time frame at any time, avoiding the typical nightly snapshots of most data warehouse applications.

These fields also have some special properties, like two predefined subfields called who and when.

Fields that can be declared as historic will be determined soon.

Internal structure

Fields into a record don't have to be in the same order they are declared. Instead, they are sorted and pushed to be correctly aligned (depending on their size) and resulting in an overall record minimum size. This means that BitMaps are packed, and small data types are used to fill holes between larger aligned fields. This could be seen as a "record compilation". You don't have to worry about, if you use the given record definitions to access data. This applies to main database records, as it does to every request you perform asking for some fields (a custom record is build every time).

Actually, the DB organizes fields in a sparse fashion to improve access time and search operations, but a user never will see them in such organization, as query results are always packed (all the record field values are contiguous).

Views

When you believe that some fields will be often used as keys for searching, declaring them as views will greatly improve that process. Views make server to arrange a copy of that field values, in a manner that makes searching in very efficient. It also makes it to consume more memory, to keep these copies, and a little more CPU, to keep them synced.

There's also an autoview feature, that arranges data as views automatically. These views are not as good as explicit views (for performance), but they don't increase the memory footprint (no extra memory is needed), and give a pretty good improvement in heavy search operations. You don't have to care about them, all the fields that can be represented in this format are automatically set as such: fixed size fields from 1 to 64 bits (including reference Ids).

Explicit views are not implemented yet (but can be declared).

Sets

There are some special field types that allow for multivalues. These are a few predefined numeric unsigned types (bSet, wSet, iSet) that can store several unique values for a single field. Number of values is limited by slot size, and depends also on type size (will hold more 8bit values than 32bit values). Values are unordered (you just can check for a value presence, but its position in list is undefined) and unique (adding a value to a set that already holds it does nothing). Maybe in some future, a new type could be defined,

something like List type, that may support repetitions and keep order, to hold lists of values.

This type is not supported yet.

Appendix 1 - Database definition file

For the very first time, database universe definition may be explained into a special text file. A special program can then create all the storage objects (in disc) for that universe, so when a BM is run it will service and declare that data structures. This program can also populate data into storage directly, a function intended for initial migration (from other sources) or even backup restoring (not on a usual working system, that should recover from a redundant BM, but for a new installation that boots from data on an independent site or system).

See the example file bundled with source distribution (record_def_base.txt).

File format is ASCII text, case insensitive (in general), allowing comments following the pound sign (#), empty lines, and MS-DOS style carriage returns. Words may be separated by spaces and TABs in any number, but usually a line break is considered to end an item definition. A section is enclosed by its reserved word (like RECORD) and its “terminating” reserved word (like /RECORD).

First, an entry must define the universe name:

```
UNIVERSE MyTestDB
```

This is compulsory, and allows you to tell the system to load this database (instead of another possible one). It also defines a scope for your data. Usually, a *global universe* is declared, with common tables (the ones that are used by any other universe), then you define your own universe (or several of them). Universes isolate data access, so a user that logged into a universe called MyDB can only see objects of that universe and, as a fallback, from the *global universe*. It is planned some sort of access control to allow a user to access several universes (but never at the same query), to improve modularity of some designs, but it still unimplemented; universe bounding happens on the connected port and can't be selected by the user (yet).

The universe fallback works like this: you ask for an object on your universe, and you get it if exists. If not, that request is tried on the *global universe* instead. If that also fails, the request finally fails.

Into every universe, information is arranged mainly in “tables”, or *records*. Some of them are expected to exist in any system, these are named with a preceding underscore. The same happens with some fields on that records. Required means that the KinDB cluster itself uses them for some system purposes, or so does a generic client interface (like KinGUI). One example is the user database, called **_user**, which is necessary for login purposes. You can expand system records with your own fields, provided you still define the required ones. This defines a simple user record:

```
RECORD _User
  ID        sID                 # Record ID
  Crea      uDTcrea             # Record creation timestamp
  _Modi     uDTmodi             # Record modification timestamp
  _Name     String8b    NomStr  # User Name
  _SurN     String8b    NomStr  # User Surname(s)
  _Boss     rsID        _User   # This user's boss (0=none)
  _Dele     rsID        _Offices # Branch or company to which this user belongs
  _Login    BitMap(1)           # 0=Can't login into system, 1=Login Enabled
  Act       BitMap(1)           # 0=Inactive (not a real user), 1=Active
  Rol       BitMap(4)           # User role (0=?, 1=admin, 2=agent, 6=accounting, 15=management)
  _lName    fText8b(16)         # Fixed size embedded text, Login Name (predefined)
```

```
  _lPasswd  fText8b(16)             # Login Password (predefined)
  _uiprof   rsID        _UIap       # Access profile for user interface for this user
/RECORD
```

This defines the required _user record (remember, names are also case insensitive), with several fields, including the required ones, and a few of my election. Bold words are reserved words. In general, a record definition is like this:

```
Record <object name>
  <Field name>      <Field Type>[Dimension]   <Referred object>
  <Field name>      <Field Type>[Dimension]   <Referred object>
/Record
```

Where field name should be a representative name for the field (system required ones use a preceding underscore, so please do not use it for your own although you're free to do so). Field type is the name of that field's type. Depending on such type, it may be followed by a dimension, enclosed in parenthesis for BitMap ot fText8b, or in brackets for an array. For fields that refer to another object (instead of hold the value itself), third parameter is the referenced object's name.

Field name may be preceded by a '+' sign, making that field an **explicit view** (a separated sorted list for its values is generated at runtime to speed searching). This is not recommended in most cases, as all the fixed size fields (for less than or equal to 64bits) are automatically pseudo-viewed, at no memory expense, so if that is fast enough for you, don't make the system to spend a lot of memory more to create real, expensive views.

Field name may also be preceded by a '~' sign, meaning that this field is the **key dimensional field** for this record. Only one can have this attribute in every record definition. Dimensional fields allow to partition a big object in rational units, based on the value of such field. It is expected to have only a few possible values (a few hundreds at most). Partitioning allows to parallelize searches, and also allows to perform them in a smaller set. Furthermore, dimensional field allows also to restrict user access to this object's values, by specifying (on the user access profile) a list of valid values for it: a user won't be able to retrieve data for the rest of dimensions. A typical dimensional field would be a 'branch' field, so a worker in one branch cannot access (even at the low level) to data belonging to other branches; data for every branch (may be) managed on separate nodes so it doesn't impact others performance.

A '*' sign may also precede the field name, meaning that this field is **historical**, that is, it can retain all the values it is assigned, and also provides information about when and who changed it. This kind of field is very useful to track the evolution of variables in time (status, price, etc), but shall be used with caution as it is also quite heavy for the system (both in storage and in searching performance). It also allows to treat the whole database as a data warehouse, as you can get a picture of every record at any arbitrary moment in time, independent of future changes to it.

Preceeding field name with '-' sign makes that field to be treated as a **unique** user-key (UUK). Any Save Request will first check this field and fail if a value is about to be duplicated: no repeated values are allowed for this field at all. Note that a value of 0 is always allowed (thus, it can be repeated). The following combinations when trying to save a record get these results (uID=field with this attribute, ID=record ID, X=any value included 0, --=not present, Vi,Vu are non-zero values):

ID	uID	
0	Vu	≠uID=Vu => Ok (create record as indicated by ID=0)

--	Vu	∄uID=Vu => create record; ∃uID=Vu => update record
X	0	ID=0 => create, ID≠0 => update (uID is simply ignored as it is 0 and that value can be repeated)
Vi	Vu	∃uID=Vu on a record that is not ID=Vi (i.e. on another record), operation is rejected (to avoid a duplicate uID value)

Since 20120608, multiple UUK fields can be declared (current limit is 8 on a record definition). When a Save Request is performed, many of them may be supplied (zero-valued or non-present ones are ignored) meaning that all of them must match to consider an existing record is addressed. So for an instance, if we have defined A, D and E as UUK fields, and we supply A and E (non-zero) on a Save Request, it is guaranteed that we can't have two records with A and E matching that values (but D is ignored and thus, we may have repeated values). Normal use means defining and using the same number of UUK fields on every Save Request, to ensure we do not have repetitions on the combination of all of them.

Any field that can be expressed as a scalar fixed-size value can be declared as unique user-key. Specifically, arrays can not be used for this purpose. Also some types are not yet supported (currently, BitMaps, sMoney and other types that have a binary representation not on 8,16,32 or 64 bits).

Other items that can be declared into the DB Universe are simply called *Object*. This is a generic variable size item storage, with the following predefined ones:

- String8b: a text strings storage object, can use the *CaseInsensitive* attribute to allow for wider matches when searching (case is not taken into account, but neither separators and other symbols). It also may use *SaveCaseInsensitive* attribute to allow only one version to be saved irrespective of its case (string is saved when it is not present -case insensitive-, and that is what it will be returned even for other versions -with any other case combination-): for an instance, if you save “David” and then “david”, only the first will be actually stored, and you will always receive it as a result (be careful to ensure the first version saved is the better: no way to change it later). Finally, *Numeric* attribute forces matching to only the numbers in the stored and requested string.

  Text strings cannot be modified (but you can create a new one). Size is often up to 255 bytes, but it may be also longer (up to 64K but usually limited to 16KB).

- Block: a binary blocks storage object, often also uses a *BaseX* attribute, to indicate the minimum unit inside (like in *Base8*: byte storage). Size may range from zero to almost 4GB, for every item. Also, items may be modified (rewritten).

Finally, you can define tasks (for the Task Module). Actually, here you only define the task “interface”, that is, the fields to communicate to it using standard DB queries. This can be seen as a Record definition, but it works in a different way: you can only use this declared fields to communicate to the task, but task can issue back (as a result) any kind of records and other objects combination, as they are defined at the response, even if they don't exist in the real system. So this may be seen more as a request API than a complete task definition.

Syntax is the same as a Record definition, only change *Record* with *Task*.

Example:

```
### Instant Messaging manager (Task)
TASK IMmgr
  Op        Byte          # Operation to perform
  uList[1]  rsID  _User   # Subscribed list of users (it actually has n-dimesion)
  uFlg[1]   Byte          # User attributes (for every one listed into uList, also n-
dimension)
/TASK
```

In this example you could request a list of subscribed users to a specific list by issuing a query like this:

```
qReq: “IMmgr.Op=1,.uList=30”

rReq: “IMmrg.Op”
```

Task would interpret Op=1 as an instruction to build a list of users, of subscribed groups that have userID=30 in its components, and send results as records with userID, Flag, groupID, nameID, and also a text strings object with the name values. Note that rReq is just informative, TM can generate whatever it decides. Op=2 could get uList as a group ID, and retrieve all that group's members, etc.

The trick here is that for most cases, a task “record” will be something like:

```
TASK aTask
  Op        Byte          # Operation to perform
  Par1      Word          # First param
  Par2      Byte          # Second param
/TASK
```

And results will be determined by the task itself, depending on the operation code.

Of course, you can define a task in the same manner you would do with a record; this is the case for most tasks that only perform *filtering*, for example, to write a new record based on partial data (or public data): you write through the task, and read raw to the IM data.

Appendix 2 – Required records and fields

The database manager, but also client interfaces, require some records (tables) and some fields inside them to exist before using the system. This is because they rely on them to perform system management. An obvious case is the *_User* record, that allows the system to identify users (and thus, to allow or disallow them to use the database itself). As a rule of thumb, all the required records an fields have an underscore preceeding their name; this is absolutely arbitrary, but helps to avoid coincidences with other user's definitions. This is a list of records, and their minimum fields the system requires (and how are used):

_User

This record is used by the access control system (at the CM) to enable a connection to perform database queries (after a successful login process). Usually, there is one record (table) at every

universe, except the *global universe*. Required fields are:

- ID (sID)

  This is the record ID, and is required that it be a 16bit short ID.

- _Modi (uDTmodi)

  This is the record last modification, and is used by caches to keep their user lists up to date.

- _Name (String8b)

  This is the user name. It is not absolutely required, but it is used by client interfaces to show the name of, at least, the logged user.

- _SurN (String8b)

  This is the user surname. Same case as above.

- _Boss (rsID → _User)

  This is the user's boss, for a users hierarchy. If not used, it may be left to 0 (root, no boss). This is used by some client interfaces that allow users to refer to other users, based on their relationships.

- _Dele (rsID → _Offices)

  This is the users office ID. Used by client interfaces to know and/or restrict user access to information based on its belonging to a company branch.

- _Login (BitMap(1))

  This indicates if user can log into the system (that is, if it has an “account” so it can perform queries), or it is just a passive one (or simply had its access revoked).

- _lName (fText8b(16))

  This is absolutely required. It is the user account name, or (case sensitive) login name. Used by the CM to enable connection to perform queries. It can be empty if _Login=0.

- _lPasswd (fText8b(16))

  Like previous case, this is absolutely required for users that can access the system, it is the account password, case sensitive too. It can be empty if _Login=0.

- _uiprof (rsID → _Uiap)

  This is a reference to the users' GUI profile. Such profile defines which applications the user has access to, and the detail of it. Required by the client interfaces.

_Uiap

This is the user interface access profiles list, and is used by the client interfaces to decide the applications a user has access to, and how it is performed. The only required field is:

- _XMLdef (Block)

  This is the profile content (XML based on the uiprofile.dtd)

_Curr

This is the currency record. It holds a list of system currencies, and is used by the DB manager to perform conversions when searching on different currencies, and by client interfaces to perform conversions and to know how to display or parse them. This record is often at the *global universe*, as it is usually common to all the other databases. Currently, it *must* be unique and global. These are the required fields and their meaning:

- _ID (Word)

  This is currency code for management (internal, arbitrary ID). For example, USD=129, EUR=169, etc.

- _cc (fText8b(3))

  This is currency international code (three capital letters: USD, EUR...)

- _exp (BitMap(3))

  This is the number of internal decimals. It must be equal or bigger than _vdp.

- _vdp (BitMap(3))

  This is the number of visible decimals.

- _xr (fp32)

  This is the current rate to the reference currency (reference currency is the one with _ID=254).

- _Modi (uDTmodi)

  This is last modification timestamp, helpful in case this record is cached.

It is also recommended to declare cs (fText8b(6)) with the UTF-8 symbol of the currency, as it is used by some client interfaces (like KinGUI).

_Files

This is a files distribution database, used by some client interfaces to manage automatic updates and generic files distribution. This includes program binaries, libraries, modules, resources (icons, window definitions...), etc. and support for different architectures. Required fields are:

- _fn (String8b → _FileStr)

  This is the original file name (and the name used to create it on the client side).

- _kind (Byte)

  This is the kind of file, and often implies the destination directory on the client side. Currently defined values are:

  - 0=?
  - 1=HLP
  - 2=DTD

    - 3=Dialog
    - 4=Language file
    - 5=img/ap
    - 6=img/sys
    - 7=img/usr
    - 8=img/wi1
    - 9=module (guimod/)
    - 10=library (lib/)
    - 11=application (bin/) (this application is considered to be running when updates may occur)
    - 12=independent application (bin/) (this application is considered to be external and not running when updates occur, so it can be updated on the fly).
- _arch (Byte)

  This is the architecture code. Currently, defined values are 0=any, 1=x86_32 Linux, 2=x86_64 Linux, 3=Win32
- _sz (Int)

  This is original file size in bytes
- _chk (Int)

  This is original file 32bit checksum (with zero padding at the end). Update clients may use this to detect a change in the file and trigger a new download of its content.
- _Dta (Block → _FileD)

  This is the original file content (referenced).
- _fTS (uDateTime)

  This is the original file timestamp.
- _Modi (uDTmodi)

  This is the last time this entry was modified, used by caches.

_FileD

This is a Base8 Block object, that holds _Files content.

_FileStr

This is a String8b object, that holds _Files texts.

_Node

This is the working nodes list, used by the access management system to athenticate connections

(at the CM). This is the very first record required, so connections can be validated to Level1 (connected, not logged in). These are the required fields:

- _IP (Int)

  This is the node base IP (Ipv4 only)
- _IPmsk (Int)

  This is the IP mask
- _CERT (fText8b(24))

  This is the node public key, in Base64 encoding.
- _Modi (uDTmodi)

  This is the last time this entry was modified.

_Offices

This is the list of company offices. It is used by _Users to place every person on the company physical hierarchy, and client interfaces may use it also to perform data access classification based on this structure. Required fields are:

- _Boss (rsID → _Offices)

  Defines the offices hierarchy (0=top).
- _Name (String8b)

  Name for the office
- _Modi (uDTmodi)

  Last time this entry was modified.

Note that _Offices and _Nodes refer to different things: while the first represents the company structure (and there may be many “virtual” offices that only organize the resources tree), the second is the list of physical computer-client locations.

_WCIdd

This is the Web Client Interface dialog definitions, and is currently unused and unsupported.

_Uidd

This is the User Interface dialog definitions, currently unused. For now, these are independent files, that are distributed through the _Files object.