Daylight Chemical Information Systems, Inc.
120 Vantis - Suite 550 - Aliso Viejo, CA 92656
tel +1 949-831-9990 - fax +1 949-831-9902 - www.daylight.com

Daylight Toolkit Programmers' Guide

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Daylight Toolkit Programmers' Guide

Daylight Version 4.9
Release Date 08/01/11

DAYLIGHT Chemical Information Systems, Inc. Aliso Viejo, CA USA

1. Introduction

Daylight's Goal
To provide the best known computer algorithms for chemical information processing to those who need them; to provide chemical information systems capable of handling all of the chemical information in the world.

Computers are used to solve many problems in chemistry, including predicting the properties of a molecule, maintaining and searching databases of chemical properties, and deducing structure from chemical properties. These tasks are challenging to the computer scientist and to the chemist.

Unfortunately, many workers in this field waste a great deal of their time creating and recreating what might be termed the *chemical information infrastructure* -- programs to read connection tables, maintain databases, depict molecules, perform sub- and superstructure searches, similarity searches, and so forth. Although all of this infrastructure is well understood and widely duplicated, it has never been made available in any reusable form. The result is that on any particular chemistry project that uses computers, the majority of time is spent recreating the infrastructure.

The Daylight Toolkit provides this chemical-information infrastructure. Using a simple set of functions based loosely on an "Object Oriented Programming" model, the Daylight Toolkit allows programmers to get directly to their unique chemical problems; they needn't be bothered with reinventing the infrastructure. With the Daylight Toolkit, programmers can get directly to the problem at hand, often saving so much time that projects can be finished in a fraction of the time that a reinvent-the-wheel approach would have required.

The Daylight Toolkit supports several popular languages, including C and FORTRAN, and is available on a variety of platforms, including several UNIX machines, PCs and Macintoshes.

1.1 Daylight Toolkit Modules

The Toolkit is divided into several modules, available separately or as packages (some require other modules to be present; for example the Toolbase and SMILES modules are required by all other modules). Below is a brief outline of each Toolkit module's features:

ToolBase This module is the foundation for all Daylight Toolkit modules. Includes Handles, streams and sequences, error functions, polymorphism, string objects, and other basic functionality.

SMILES (Simplified Molecule Input Line Entry System) SMILES input and output; unique-SMILES generation; molecular connection tables; isomerism; chirality; addition, deletion, and modification of atoms and bonds; substructure objects (but not substructure searching -- see SMARTS). (See Chapter 2 for a complete description of SMILES).

SMARTS (SMILES Arbitrary Target Specification) A substructure search system utilizing SMARTS, an extension of SMILES which allows chemically meaningful expressions to be constructed.

Fingerprints Generating molecular "fingerprints", characteristic arrays of bits that allow high-speed screening for substructure- search systems, and similarity metrics for molecules.

Depict 2-D schematic representations (depictions) of molecules; 3- D conformations; depiction modification; atom, bond, and whole- depiction labels; rotations.

Thor (THesaurus Oriented Retrieval) Chemical Databases using a thesaurus-like approach; allows high-speed storage and retrieval of chemical information using ambiguous or inexact identifiers, synonyms, trade names, and so forth.

Merlin High-speed in-memory searching of chemical structure and chemical information.

1.2 Audience and Background

This document is intended for programmers who will be incorporating the Daylight Toolkit into their own programs (applications). Because of the diversity of backgrounds expected in such an audience (from Computer Scientists to Chemists to System Administrators), we try to err on the side of being verbose.

Experience with computer programming and chemistry is expected; in particular, you must be familiar with your application language (FORTRAN, C, Pascal, etc.), and with basic chemical nomenclature. Familiarity with data structures such as hash tables is helpful but not necessary, as is an acquaintance with the concept of "Object Oriented Programming".

1.3 Other References

The Daylight Chemical Information Systems: Theory of Operation manual is background for this manual. A thorough understanding of its contents is necessary before this material will make sense.

This manual is intended to serve as a tutorial introduction to toolkit programming. The on-line manual pages serve as the authoritative reference for toolkit functionality and behavior.

This document is a companion to the Daylight Toolkit Reference Card, which contains exact specifications for the functions available in the Daylight Toolkit, and comes in language-specific versions (C, FORTRAN, etc.).

The Daylight Toolkit also comes with a number of ready-to-compile example programs in the "contrib" directory. These can be extremely useful as a starting point in working with the Toolkit. We suggest that you glance through the examples before serious perusal of this manual to get an idea of what's there. The examples may clarify many of the explanations given here.

1.4 Conventions

Because the Daylight Toolkit is designed to work with a variety of languages, we use a generic or "function prototype" technique to describe each function. For example, consider the following function prototype:

```
    dt_type(object ob) => integer
```

It translates to the following:

```
    C Prototype:         int dt_type(dt_Handle ob)

    FORTRAN Prototype:   integer function dt_type(ob)
                         integer ob
```

> **NOTE:** The actual function calls for a given language many be *significantly different* from the "prototypes" shown in this manual. Consult the online man-pages for exact function specifications.

In particular, strings are represented differently in most languages. For example, the function prototype:

```
    dt_stringvalue(Handle ob) => string s
```

would translate to the following C function:

```
    char *dt_stringvalue(int *len, dt_Handle ob)
```

Notice that the actual function doesn't even have the same number of parameters as the function prototype! In general, you should read the manual to get the function's description, then refer to the Daylight Toolkit Reference Card for the function's exact syntax.

Descriptions of functions that return strings often refer to the *invalid string*, which is often returned when a function that returns a string detects an error. The specific definition of the *invalid string* is language dependent.

1.5 Compiling and Linking

As a Daylight Toolkit programmer, you will compile source code to object code and link object code to an executable binary using the Daylight Toolkit and operating system libraries. The following information shows the syntax for building a Daylight Toolkit program.

1.5.1 Compiling

Let's say you have the following source code in a file named smiles.c:

```
#include <stdio.h>
#include <string.h>
#include "dt_smiles.h"

void main(int argc, char **argv) {
    dt_Handle mol;

    if (2 > argc)
        printf("Usage: %s <SMILES>",argv[0]);
    else if (NULL_OB == (mol=dt_smilin(strlen(argv[1]),argv[1])))
        printf("SMILES is not valid.\n");
    else
        printf("SMILES is valid.\n");
    dt_dealloc(mol);
}
```

The syntax for compiling source code is:

`compiler [ options ] file`

The *compiler* typically is the operating system standard C-code (**cc**) or FORTRAN (**f77**) compiler, or the GNU project C and C++ (**gcc**) or FORTRAN (**g77**) compiler.

The *option* to compile source code to object code is **-c** and the *option* to specify the directory location of Daylight "#include" files is **-I$DY_ROOT/include**.

The *file* is **smiles.c**.

The following compiles the smiles.c source code to object code file and producing an object code file named **smiles.o**.

```
cc -c -I$DY_ROOT/include smiles.c
```

1.5.2 Linking

Now, let's link the object code to an executable binary. The syntax for linking source code is:

`compiler [ options ] file toolkits [ libraries ]`

The *compiler* is the same as before.

The *option* to link object code to an executable binary is **-o <*filename*>**, e.g., **-o smiles**, and the *option* to specify the directory location of the Daylight Toolkit libraries is **-L$(DY_ROOT)/lib**.

The *file* is **smiles.o**.

The *toolkits* define Daylight functions. In this case, the SMILES Toolkit is required by **smiles.c** (which calls dt_smilin(3) and dt_dealloc(3)), and is specified by **-ldt_smiles**.

The *libraries* are not needed, as no operating system functions are used in **smiles.c**.

The following links the smiles.o object code to the the SMILES Toolkit, producing an executable binary file named **smiles**.

```
cc -o smiles -L$(DY_ROOT)/lib smiles.o -ldt_smiles
```

Alternative, you may combine compile and link command, e.g.,

```
cc -o smiles -I$DY_ROOT/include -L$DY_ROOT/lib smiles.c -ldt_smiles
```

1.5.3 Toolkit Libraries

1.5.4 Advanced Programming

X libraries is **-L$(XVIEW_LIB) -L$(X_LIB)**. The definition of **XVIEW_LIB** and **X_LIB** is operating system dependent and shown in the table below:

Platform	XVIEW_LIB	X_LIB

Red Hat Linux/usr/openwin/lib/usr/X11/libSGI Irix 32-bit/usr/local/openwin/lib/usr/lib32SGI Irix 64-bit/usr/local/openwin/lib/usr/lib64SUN Solaris/usr/openwin/lib/usr/X/lib

The *files* are the object code, which may be several files. In this case, there's one file, **smiles.o**. The *toolkits* required will depend on what parts of the Daylight Toolkit you use in your program. You can determine which toolkits are required from the "Library Linkage" section of the Daylight Toolkit Functions manual pages or visiting the Daylight website at http://www.daylight.com/dayhtml/doc/man/man3/index.html. The syntax for linking to a library is **-l<*library*>** and multiple libraries must be linked in a specific order on IRIX and Solaris (Linux excluded). Below is a list of Daylight Toolkits, library syntax and dependancies in the order of required linkage.

Toolkit	Syntax (DY_LIB)	Dependancies
SMILES	-ldt_smiles	none
SMARTS	-ldt_smarts	-ldt_smiles
Fingerprint	-ldt_finger	-ldt_smiles
Reaction	-ldt_smiles	none
Reaction w/ Transforms	-ldt_smarts	-ldt_smiles
Thor	-ldt_thor -ldt_ipcx	-ldt_smiles
Merlin	-ldt_merlin -ldt_ipcx	-ldt_smarts -ldt_smiles
Rubicon	-ldc_rube	-ldt_depict -ldt_smarts -ldt_smiles
Program Object	-ldt_progob	-ldt_smiles
Grins Widget	-ldw_xvgrins	none
TDT Widget	-ldw_xvtdt	none
Basic Widget	-ldw_xview	none
Depict	-ldt_depict	none
"contrib" Depict	-ldl_xview or -ldl_stubs	none
"contrib	-ldu	none
none (for database definitions)	-ldt_datatype	none
none (for XView applications)	-ldt_apputils	none

Finally, the *libraries* required will depend on your use of X graphics and operating system routines. Some Daylight Toolkits require X libraries (Depict Toolkit) and all require operating system libraries. Similar to Toolkit libraries, the syntax for linking to a library is **-l<*library*>** and must be linked in a specific order. Below is a list of X Graphics and operating systems and library syntax in the order of required linkage.

X Graphics	Syntax (GFX_LIB)
XView	-lxview -lolgx -lX11 -lXext
Operating System	**Syntax (OS_LIB)**
Red Hat Linux	-lnsl -ldl -lm

SGI Irix 32-bit	-lsocket -lnsl -lw -ldl -lm -lmalloc
SGI Irix 64-bit	-lsocket -lw -lm -lmalloc
SUN Solaris	-lsocket -lnsl -lw -ldl -lintl -lm -lmalloc

Note: XView graphics is not available on SGI Irix 64-bit systems. Now we have all the information we need to link together a Daylight Toolkit program. If you programmed use of all Daylight functions, the syntax for linking a HelloWorld C program on a 32-bit SGI Irix would be: cc -o HelloWorld HelloWorld.c $(DY_ROOT/lib) $(XVIEW_LIB) $(X_LIB) $(DY_LIB) $(GFX_LIB) $(OS_LIB) ${DY_ROOT}/contrib/lib/libdl_stubs

2. Basics: Daylight Toolkit Objects

2.1 Introduction to Objects

The Daylight Toolkit uses an object mechanism to simplify the task of programming for chemistry. We begin with a short example that illustrates the object-aspects of the Daylight Toolkit (many other examples come on the tapes with the Toolkits).

```
  /*-----------------------------------------------------
   *  thorload.c -- a simple program to load data into
   *  a THOR database.
   *-----------------------------------------------------*/

  #include <dt_smiles.h>
  #include <dt_thor.h>

  main(int argc, char **argv)
  {
    dt_Handle server, db, tdt;
    char dbname, servername;
    char *tdtbuf;
    int   tdtbufsize, tdtlen, isnew;

    /**** Specify the server (machine) and database name ***/
    servername = "my_machine_name";
    database = "medchem02demo";

    /**** Connect to server and open database ****/
    server = dt_thor_server(strlen(servername), servername,
                   4, "thor", 4, "thor", 0, "");
    db     = dt_open(server, strlen(dbname), dbname, 1, "w", 0, "", &isnew);

    /**** Load data until input is exhausted ****/
    while (1 == du_readtdt(stdin, &tdtlen, &tdtbuf, &tdtbufsize)) {
    dttdhor= str2tdt(db, tdtlen, tdtbuf, 1);
dt_thor_tdtput(tdt, 0);
    }
    dt_dealloc(server);
  }
```

The important features of this example are:

- The variables **server**, **db** and **tdt** are identifiers, or handles, that your code and the Daylight Toolkit use to refer to an *object* (in this case, Thor server, Thor database, and Thor datatree objects, respectively).

- The handles themselves contain no information - they are not pointers to complex structures.
- The Daylight Toolkit manages the objects for you - you need not be concerned with details of how the Toolkit represents the molecule or depiction.
- Error handling is simplified: the NULL object (returned when errors are detected) is a valid handle that refers to nothing. In the above example, the Daylight Toolkit will not generate fatal errors even when the server can't be reached or the database can't be opened; it simply will not load anything. Functions (not illustrated) are provided to retrieve error messages from the Toolkit.
- Because objects are managed by the Daylight Toolkit, the interface to various programming languages is straightforward: the Daylight Toolkit works equally well with C, FORTRAN, Pascal, or LISP.

Many programmers will recognize the similarity of this approach to Object-Oriented Programming (OOP). Although many of the ideas described here are borrowed from OOP, the Daylight Toolkit is not as complete or complicated as a true OOP system. However,the Toolkit uses the following key OOP concepts:

You deal with objects:
: Everything you work with, such as molecules, depictions, databases and THOR Data Trees, is an object.

Objects are referenced by their handles:
: An object is referred to by its handle; something the Toolkit gives you when you create the object. This handle is typically an arbitrary 32-bit integer, but even this level of detail is irrelevant: it does not matter to you what a handle is so long as you use it correctly.

Handles are opaque:
: An object's handle is all you know about directly. The handle is opaque -- you can't see what is inside the object it represents.

The Toolkit uses a strict functional interface:
: You never work on data structures or "common blocks". Instead, you call Toolkit functions to create, modify, use, and destroy Toolkit objects.

Objects are self-describing:
: Each object "knows" what it is. Many Toolkit functions will take a variety of different object types (they are polymorphic; see the chapter entitled POLYMORPHIC FUNCTIONS); the function "asks" the object what type it is and performs the appropriate action.

2.2 Handles

As noted above, handles are the Toolkit's "name" for each object that it creates, and the handle is the only thing your application program knows directly about the object's representation. Because objects are opaque, it is irrelevant to you what a handle actually represents (in fact, different versions of the Toolkit use different methods to assign handles to objects).

Although handles are opaque, they have several properties that are important to the application programmer. These properties are the only ones that the Toolkit guarantees:

Uniqueness: Each handle is guaranteed to be unique at all times:

- If two handles are equal, they refer to the same object.
- If two handles are not equal, they refer to different objects.

Note that uniqueness is not guaranteed over time: the Toolkit may re-use a handle if the original object it represents is discarded.

Revocation: Some toolkit functions cause previously-returned handles to become invalid. For example, the handle for an object becomes invalid if the object is removed from the system with dt_dealloc(). A handle that has become invalid in this way is said to have been *revoked*. Generally speaking, all operations on revoked handles produce undefined results. It is up to the application programmer to guarantee that revoked handles are not used. For functions that cause revocations, the specific description of each function in the Daylight Programmer's Reference Manual will say exactly which handles are revoked.

Vigilance: To assist programmers during code development, "vigilant" versions of the Daylight Toolkit are available. These versions may be able to detect the use of an invalid handle. In other words, some toolkit implementations do define a behavior when an operation is applied to a revoked handle. In such vigilant versions, passing a revoked handle to a toolkit function will cause an error return. For extra help in detecting errors, a function named dt_invalid() may be used to test the validity of a handle; it is explained more fully in the chapter entitled POLYMORPHIC FUNCTIONS. A second vigilance function, _dt_vh_stop_here(), is provided for use with a debugger. The Toolkit calls this function when an invalid handle is detected.

2.3 Object Types

The Daylight Toolkit supports a small number of object types. These are divided into several sections, corresponding to the Toolkit's parts (e.g. SMILES, Depict, THOR, etc.). Each of these object types is explained in more detail in the chapter for that section of the Daylight Toolkit; here we give an abbreviated list as an introduction to the object-type concept.

General:

stream	An ordered enumeration of objects from a base object
sequence	an ordered sequence of objects of any type

SMILES:

molecule	a molecule structure
atom	an atom in a molecule
bond	a bond in a molecule
cycle	a cycle in a molecule

Depict:

depiction	a 2-d representation of a molecule
conformation	a 3-d representation of a molecule

THOR

server	a connection to a THOR server process

database	a database of chemical information
datatree	a single entry from a database
dataitem	a datum from a datatree

Each of the above object types is represented by a symbolic constant:

Object Type	Symbolic Name
server	TYP_SERVER
stream	TYP_STREAM
molecule	TYP_MOLECULE

and so forth. The exact symbolic names for each object type can be found in the Daylight Programmer's Reference Manual.

There are two "pseudo object" types: TYP_ANY and TYP_INVALID. The pseudo object type TYP_ANY is used when any object is acceptable. Since it is a pseudo object type, there are no actual objects of type TYP_ANY. Similarly, TYP_INVALID may be returned by functions to indicate that the specified object is unknown or incorrect. There are no actual objects of type TYP_INVALID.

2.4 The NULL_OB Handle

One special handle value is used to represent "nothing"; it indicates that no object is present. It is called the null object, and its handle is represented by the symbolic constant NULL_OB. A handle whose value is NULL_OB is a valid handle, but it does not refer to any object and it has no type.

NULL_OB plays a special role in the Daylight Toolkit: Functions that return objects will return NULL_OB if an error occurs, and functions that take object parameters will accept NULL_OB as a valid handle (they ignore it and do nothing). This means that error management in applications that use the Toolkit is somewhat simplified -- in many cases the handle returned by one function can be safely passed to the next function whether the first function failed (returned NULL_OB) or succeeded (returned a handle to a real object). It is safe to pass NULL_OB anywhere a handle is expected. See the chapter on error handling for more discussion of this topic.

NOTE: In current implementations, NULL_OB is defined to be zero. However, there is no guarantee that this will always be the case. Application programs should explicitly compare for equality or inequality to NULL_OB rather than using constructs like "if (!my_handle) ...". Programs that assume NULL_OB is zero are explicitly non-portable.

2.5 Daylight Version Handling

The Daylight Toolkit has both Runtime version handling and Compile time version handling. The runtime version handling can be used in the user code to show which version of the runtime libraries are currently being used. The user code can compare the version number to the current Daylight release version and if it is different print an error message describing version inconsistency along with a suggestion to check LD_LIBRARY_PATH which tells the code which runtime libraries to use.

The runtime version and creation date can be accessed with the dt_info() function. If the dt_info() function is called with the "toolkit_version" parameter with the runtime libraries made with version 4.81 or later it will return a version number. Any libraries made prior to 4.81 will return NULL.

The compile time versions refer to when the entire toolkit was compiled. These versions are described in dt_smiles.h with DX_TOOLKIT_VERSION and DX_TOOLKIT_DATE. These are also the versions numbers and dates that are referenced in the man pages and other Daylight documentation.

The user can use DX_TOOLKIT_DATE and DX_TOOLKIT_VERSION to ensure that they are compiling their code with the correct runtime libraries.

Example of using runtime and compile time versions.

```
int main()
{
...
rver = dt_info(&rlen, NULL_OB, "toolkit_version");
if (rver == NULL)
     printf(stderr, "WARNING: you're using an older (pre-4.8) "
     "toolkit runtime library, check LD_LIBRARY_PATH");

else if (0 != strncmp(rver, DX_TOOLKIT_VERSION, rlen)))
     printf(stderr, "You compiled this program with "
     "version %s but are running it against the "
     "%.*s toolkit runtime library.\n", DX_TOOLKIT_VERSION,
     rlen, rver);
}
```

3. Basics: Polymorphic Functions

3.1 Polymorphism

There are many functions, such as counting, copying, deallocating, and naming, that can be applied to several different types of objects. We refer to these functions as polymorphic.

The idea of a polymorphic function might seem foreign at first, but it is actually quite familiar to all programmers. Take, for example, the "*" operator in FORTRAN. When applied to two numbers, we expect it to cause the two numbers to be multiplied. However, on closer inspection, the "*" operator turns out to be polymorphic: it can be applied to integers, single-precision floating-point numbers, double- precision floating-point numbers, and complex numbers.

The difference between the FORTRAN style of polymorphism and that employed by the Daylight Toolkit is only that the nature of the operation is determined at run time rather than at compile time. That is, the FORTRAN compiler looks at the operands and decides which of several functions to apply, then generates the appropriate code; at run-time the decision as to which function to apply has already been made. In the Daylight Toolkit, a dispatch function examines the object of interest and decides "on the spot" (i.e. at run time) which function to apply.

Not all polymorphic functions can be applied to all objects. The following two sections respectively describe "generic" polymorphic functions (those that apply to all objects) and "semi-generic" polymorphic objects (those that could apply to more than one object but not to all objects).

NOTE: The specific behavior of polymorhic functions when given different object types is rigorously defined in the reference manual pages.

As a simple example of the power of polymorphism, the following function accepts any object and prints out all of its string value(s):

```
     dt_Integer dump_strings(dt_Handle ob) {
         dt_Handle m, d;
         dt_String line;
         dt_Integer len;

         /*  Get the stringvalue */
         line = dt_stringvalue(&len, ob);

         /*  If the object has one, print it. */
         if (line != NULL)
           fprintf(stderr, "Stringvalue is: %.*s\n", len, line);

         /* Check to see if the object is a stream or sequence.  If
            so, examine the members also.*/
         if ((dt_type(ob) == TYP_STREAM) ||
             dt_type(ob) == TYP_SEQUENCE))
           {
   dt_reset(ob);
           while (NULL_OB != (m = dt_next(ob)) && !dt_atend(ob))
             dump_string(m);  /* Call recursively for each member. */
           }
         return (TRUE);
     }
```

The important features of this example are:

- The function need not know in advance the type of object which may be used in this function. The only exception is in cases where special processing is desired (here, for streams and sequences).
- If dt_stringvalue() fails, we don't do anything special. It simply means that the given object doesn't have a string value, or doesn't respond to the dt_stringvalue() function. In either event, we continue.

3.2 Generic Functions

The following work on all Daylight Toolkit objects.

`dt_adjunct(Handle ob) => object`
: Retrieve the adjunct object associated with `ob` (see dt_setadjunct()).

`dt_invalid(Handle ob) => boolean`
: If the Daylight Toolkit is of the "vigilant" type and can determine that the given handle `ob` is invalid, return TRUE. Otherwise return FALSE.

`dt_setadjunct(Handle ob, object adjunct_ob) => object`
: Makes `adjunct_ob` the adjunct of `ob` -- a simple mechanism to let one object "point" to another.

`dt_type(Handle ob) => integer`
: Return the type of the given object, represented as an integer.

`dt_typename(Handle ob) => string`
: Return a string naming the type of the given object, e.g. `"molecule"` for a molecule object.

3.3 Semi-Generic Functions

The following functions are generic in that they apply to more than one object type, but there may be object types to which they do not apply.

`dt_add(Handle set, Handle object) => boolean`
: Adds `object` to set.

`dt_base(Handle ob) => object`
: Returns the base object -- the object from which `ob` was derived. Examples of objects that have a base object are: depictions and conformations (base object is a molecule); and streams (base objects are molecules, THOR data trees, etc.).

`dt_copy(Handle ob) => object`
: Returns a handle for a copy of the given object. A copy of an object shares no structure with the original. A copy of an object is guaranteed to behave exactly like the original in every respect.

`dt_count(Handle ob, integer typeval) => integer`
: Counts and returns the number of objects of the specified type within or associated with the object ob.

`dt_dealloc(Handle ob) => boolean`
: The given object is removed from the system and its handle is revoked. Frees all resources used by the object (memory, open files, etc.). Once revoked, a handle must not be used; doing so has undefined results, which may include "crashes" of the Toolkit.

`dt_info(Handle ob, string whatinfo) => string`
: Return information about an object. Many objects have special properties in the sense that they are not set by Toolkit functions, but rather arise from external sources. An example is a THOR database: a call to dt_info(db_handle, "users") will return a string containing the names of all other users who currently have the database open. Using dt_info with a NULL_OB will return information about the runtime library that is currently being used. Using "creation_date" as the string will return the creation date of the runtime library and using "toolkit_version" will return the toolkit version number.

`dt_member(Handle set, Handle object) => boolean`
: Returns TRUE if the given object is a member of the set.

`dt_molecule(Handle ob) => molecule`
: Some objects (e.g. THOR Data Trees) have a "hidden" molecule object associated with them. It is often convenient to use this molecule object rather than re-creating it; for example, when you want a unique SMILES (dt_cansmiles()) for the object of interest. This function will return the hidden molecule's handle.

`dt_parent(Handle ob) => object`
: Returns the parent of the specified object. Examples of objects that have parents are: atoms and bonds (parent is a molecule object); THOR data tree (parent is the database from which the data was retrieved).

`dt_remove(Handle set, Handle object) => boolean`
: Removes `object` from the `set`.

`dt_setstringvalue(Handle strobj, string str) => boolean`
: Changes an object's contents to the specified string. Several objects, include string objects, THOR datatree objects, and Fingerprint objects, have a string value that can be set by this function.

  The object maintains its own copy of `str`, so the contents of str need not be maintained after calling this function. Note that not all objects that return a string (see dt_stringvalue()) allow you to set the string value; in some cases the string value is derived from other properties.

`dt_stringvalue(Handle ob) => string`
: Returns the string value of the object. Many objects, include string objects, THOR objects, and Fingerprint objects, have a string value that can be accessed by this function.

The string returned by this function is "owned" by the Toolkit, and should not be modified in any way by the application. For example, attempting to directly overwrite the contents of a string object is an error -- although it may work in one implementation or with one particular language, it is an unsupported operation and may fail in future releases of the Toolkit or with compilers on different operating systems. One should always use dt_setstringvalue() to change an object's contents.

`dt_stream(Handle ob, integer typecode) => stream`

This polymorphic function is described in detail in the chapter on Stream and Sequence objects.

4. Error handling

4.1 Introduction

Using an object-oriented approach to programming interfaces can make error-handling much simpler. For example, the NULL_OB is used extensively as a returned object under conditions where a function does not return a 'valid' object. Initially, one might reasonably think of the return of the null object as an indication of failure of the function, or a flag that an invalid operation was attempted. Using a traditional procedural programming approach, this is a perfectly normal way to think about the NULL_OB.

The departure from traditional error handling comes when one examines the NULL_OB itself. The NULL_OB is a perfectly valid object in the toolkit (as valid as any other object). The NULL_OB is defined to have all of the properties of 'normal' objects, and can be passed legally to any toolkit function as an object parameter. This is completely different from the error-handling techniques used for procedural programming. In procedural programming, errors must be trapped immediately after they occur, because functions downstream of the error can exhibit undefined or invalid behaviors.

Herein lies one of the strengths of the object-oriented programming approach. Error trapping need not be performed after every function, but only when errors that significantly affect the operation of the program may occur. That is, several toolkit functions can be considered a functional block from an error-trapping perspective. The advantage is that normal processing incurs less overhead for error detection. The overhead is only incurred when necessary and most of the overhead is outside of the normal stream of processing.

The following trivial example illustrates the point. It only checks for abnormal conditions after executing several dependent toolkit functions:

```
     #include "dt_smiles.h"

     dt_Handle make_integer(dt_Integer value, dt_Handle itsadjunct) {
         dt_Handle intob, rcob;
         dt_Boolean rc;

         /*  Do the work */
         intob = dt_alloc_integer();
         rc = dt_setintegervalue(intob, value);
         rcob = dt_setadjunct(intob, itsadjunct;

         /*  If everything worked, return. */
         if (rc && (rcob != NULL_OB))
           return (rcob);   /* m is the same as rcob */

 dt_dealloc(intob);
```

```
        /* What type of error was found??.*/
        if (intob == NULL_OB) {
          fprintf(stderr, "Couldn't allocate object.\n");
          return (NULL_OB);
        }
        if (rc == FALSE) {
          fprintf(stderr, "Couldn't set integer value.\n");
          return (NULL_OB);
        }
        if (rcob == NULL_OB) {
          fprintf(stderr, "Couldn't set adjunct.\n");
          return (NULL_OB);
        }
    }
```

The important features of this example are:

- This example is overkill. In most cases, it would be preferable to simply free the integer object and return NULL_OB if any failure occured. We can successfully ignore all of the errors that may occur.
- dt_dealloc() is invoked to try to free the integer object. Note that we don't bother to check or keep track of whether or not any object was successfully allocated. If there isn't any object to deallocate, the dt_dealloc() doesn't do anything.

4.2 General approach

There are two fundamental principles which dictate the Daylight Toolkit approach to function operation. They are that:

- all toolkit functions are valid for all object types,
- all object properties are defined for all object types.

The implication of these principles is that by definition, most Daylight Toolkit functions always succeed. This does not mean that all combinations of functions and objects make sense, but there is rarely a danger of causing an error or corrupting the Daylight Toolkit by calling a function with any arbitrary object. Furthermore, the results of all functions are rigorously defined for all objects. In most cases, the most dire consequence of an inappropriate object/function combination is that the function is ignored.

4.3 Function types

The Daylight Toolkit functions can be divided into classes based on the type of operation that they are performing, and the returned type. There are three types of operations performed by Daylight Toolkit functions:

1. Functions which create objects
2. Functions which get the properties of objects
3. Functions which modify the properties of objects.

4.3.1 Functions which create objects

These functions always succeed and return an object. The object will be the NULL_OB if the attempted creation of an object is not appropriate for the given combination of arguements.

Examples: dt_alloc_mol(), dt_open(), dt_addatom().

4.3.2 Functions which get the properties of objects

These functions always succeed (by definition), and will return the defined property for the object. Since all properties are defined for all objects, the programmer is responsible for the object types passed to functions. For example, dt_aromatic(server) is defined as FALSE, and the programmer is responsible for recognizing this 'nonsensical' case and avoiding it in applications. The main way to do this is to define the valid object types for user-written polymorphic functions and check that the parameter types are valid. This level of rigor is typically only necessary for debugging and for some special applciations where it is critical to avoid ambiguities. For example:

```
#define NOT_A_RING -1
#define IS_AROMATIC_RING 1
#define NOT_AROMATIC_RING 0

dt_Integer is_ring_aromatic(dt_Handle object)
{
  dt_Boolean rc;
  if (dt_type(object) != TYP_CYCLE)
    return NOT_A_RING;
  if (dt_aromatic(object))
    return IS_AROMATIC_RING;
  else
    return NOT_AROMATIC_RING;
}
```

Without the prior object-type checking, the function `is_ring_aromatic()` would return NOT_AROMATIC_RING if given a server, database, non-aromatic bond, etc. The function does not fail if the object-type checking is not performed, but the results may not be as intended.

Note also that functions which return streams or sequences are considered functions which return object properties. Although they typically create a new stream or sequence object, the stream or sequence contains a set of properties of the object given as a parameter. None of the functions in this class will return an empty stream or sequence; they will either return a stream or sequence with one or more members, or they will return the NULL_OB.

Examples: dt_aromatic(), dt_fp_tanimoto(), dt_symbol(), dt_invalid(), dt_mer_sortapplies(), dt_xatom(), dt_getdatabases(), dt_charge().

4.3.3 Functions which modify the properties of objects

These functions are defined to be valid for all object types but may or may not succeed. In all but a few cases, this class of function returns a boolean. These functions will return TRUE if the operation succeeded and if the property was set and FALSE if the property was not set. As with functions which get the properties of objects, checking the type of the object parameters is the sole required safeguard.

Examples: dt_calcxy(), dt_rotate(), dt_fp_fold(), dt_setaromatic(), dt_add().

4.3.4. Exceptions

Several functions both modify and return properties. (eg. dt_fp_setminsize()) These typically take a property value and return the property value after the new value is applied. If the new value is not appropriate and the modification of the property fails, then the property value returned is the value that had been set prior to the function.

Merlin hitlist operations can both modify and return properties of hitlist objects. These functions typically perform an operation, and then return the size of the hitlist (which may have been altered by the function).

Functions operating on sets modify the contents, but return the object that was modified, as opposed to a boolean for success or failure (dt_add(), dt_append(), dt_insert(), dt_setadjunct()). In some cases, a duplicate object is created and added to the set. The returned object is newly created member of the set. This is the case for paths in pathsets. Otherwise, the functions return the handle of the object to which the new object was added.

Functions which return streams or sequences appear ambiguous. Some cases are clearly returning properties (dt_getdatabases()), others less so (dt_match()). In an abstract sense, one can think of these as cases of 'lazy' evaluation. In the case of dt_match(), we could argue that the resulting pathset is a property of the molecule and pattern, even though the pathset is not evaluated until the dt_match() function is called. (A stretch, perhaps).

4.4 Function return types

In addition to considering the operation which each function performs, we can consider the specific returned type from each function. There are five types of returned values from toolkit functions:

- dt_Boolean
- dt_Handle
- dt_Integer
- dt_Real
- dt_String

Each returned type has a specific value which indicates an abnormal condition.

4.4.1 Functions which return dt_Boolean

These functions are one of two cases: functions returning boolean properties, and functions returning success/failure when setting a property. There are quite a few boolean properties of objects (such as aromatic, atstart, mod_is_on). For each of these functions, the man page enumerates all of the object types which may have a TRUE value for each property. All other objects for a given property have been defined such that the property is always FALSE.

All of the functions which are used to set object properies return TRUE or FALSE depending on the success or failure of the operation. The man page enumerates the objects whose properties may be modified for each function of this type.

4.4.2 Functions which return dt_Integer

These functions all return integer properties of an object or are hybrid functions that modify and then return a property of an object. The man page erumerates all of the object types which will return useful integer properties. By definition, the integer properties of all other object types are defined as -1.

Exceptions include merlin hitlist operations, which modify a hitlist and return its length, dt_ping(), which performs an 'external' operation (it does not operate directly on objects), the (obsolete) fingerprint functions to set global options (e.g. dt_fp_setminsize()), which set a value and return the new value, and dt_thor_tdtput() which returns the success or failure of the operation, but uses an integer because of the need for success / failure / timestamp-out-of-date.

4.4.3 Functions which return dt_Real

These functions all return a real property for an object. For each of these functions, the man page enumerates the object types which have modifiable real properties. The real properties for all other object types are defined to be -1.0.

The one exception is dt_fp_setmindensity(), which modifies a property and then returns the new value of that property.

4.4.4 Functions which return dt_String

All of these functions return the string property of an object. For each of these functions, the man page enumerates the object types which have modifiable string properties. The string properties for all other object types are defined as the *invalid string*.

4.4.5 Functions which return dt_Handle

These functions either create new objects or return the object properties of objects. For each of the functions which get the object properties of objects, the man page enumerates the object types which have modifiable object properties. The object properties for all other objects are defined as the NULL_OB.

The exceptions are functions which modify sets(dt_add(), dt_insert(), dt_append()). They modify the set, and return either the given set or a copy of the object which was added to the set depending on the context.

4.5 Error message facilities

Various Daylight Toolkit operations can result in errors. The errors typically encountered are a direct result of external interactions of the Toolkit of three general types: failures of input of external data as part of the creation of objects, failures of communications with external resources (servers, toolkits, databases), and exhaustion of resources available to the Toolkit (out of memory).

Failures of parsing of external data (eg. parsing SMILES for dt_smilin) typically result in diagnostic messages which allow debugging the external data expression. Failure of external communication and exhaustion of Toolkit resources are provided primarily for graceful termination of application. The Daylight Toolkit provides several functions to access and clear the Toolkit- provided diagnostic messages, and to add your own diagnostic messages to the Toolkit's queue.

The error-handling functions maintain a queue of approximately 200 error messages. If this error queue overflows, the last message in the queue is lost and the newest message replaces it.

There are several levels of error message:

Error Level	Explanation
DX_ERR_NONE	No error.
DX_ERR_NOTE	Something that might or might not be of interest, but not an error.
DX_ERR_WARNING	Something abnormal that may require attention

DX_ERR_ERROR	The requested operation could not be carried out
DX_ERR_FATAL	A serious error was detected; the Toolkit cannot continue.

A "fatal" error does not actually cause the application program to quit; you have time to clean up, close files and warn the user that something serious has occured. However, once a fatal error has occurred, the Toolkit's ability to continue correctly is doubtful.

Note also that a fatal error may not be reported correctly since many fatal errors involve a memory allocation failure. The error- reporting functions may fail to allocate memory needed to record the error messages, resulting in lost error messages.

`dt_errorclear() => void`
: Discard all error messages and clear the error queue.

`dt_errorworst() => integer`
: Returns the error level of the worst error recorded by dt_errorsave() since the last call to dt_errorclear() or since the application program started. A return value of DX_ERR_NONE indicates that no errors have been recorded.

`dt_errorsave(string func_name, integer level, string message) => integer`
: Store the error message at level in the error queue. The parameter `func_name` is also stored; typically it is the name of the function in which the error was detected. `func_name` is printed in parentheses after the error message by dt_errors().

`dt_errors(integer level) => Handle sequence`
: Return all errors of severity level or worse (higher); if level is zero, all notes, errors and warnings are returned. The returned sequence is a sequence of string objects, in generation order.

  The sequence and string objects are newly-allocated copies of the error queue; it is the responsibility of the calling function to eventually deallocate the sequence object and all of the string objects it contains.

`dt_smilinerrors() => Handle sequence`
: Like dt_errors(), above, but returns errors related to SMILES parsing. Prior to version 4.91 this was handled separately than the regular error queue. Starting with version 4.91 this function is identical to `dt_error(DX_ERR_NOTE)`.

5. Basics: String and Number Objects

As a convenience, the Daylight Toolkits provide object to hold "ordinary" data -- strings, integers, and floating-point numbers. There is nothing fancy about these objects -- they simply hold a string or numeric value. However, the ability to represent strings and numbers as Toolkit objects makes it possible to attach all kinds of information (e.g. atomic properties, labels, etc.) to other objects, such as atoms, bonds, and molecules. The functions dt_adjunct() and dt_setadjunct() are particularly useful with these simple objects.

5.1 String Objects

A string object is a simple Toolkit object that holds the characters in a string, and the string's length. The Toolkit's concept of a "string" is a series of arbitrary 8-bit values and a length. Note, in particular, that the string is not assumed to be null-terminated, padded with blanks, or other application-language-dependent standards. For example, it is permissible to store arbitrary, or "binary" data in a string object as long as it can be represented as a series of bytes with no machine-dependent word-alignment assumptions.

(Note: Many Toolkit functions return strings. Don't confuse this with functions that return string objects. If a function returns a string object, then the function's return type will be a Handle.)

There is only one function specific to string objects:

`dt_alloc_string(string s) => ob`
: Creates a string object whose contents are identical to s. The string object contains a copy of s rather than a pointer to s, so the application program can discard the original string after calling this function. If s is the invalid string, the string object's contents will be the invalid string.

A string-object's contents can also be modified via the function dt_setstringvalue(), and retrieved with the function dt_stringvalue(); both are discussed in the previous section.

5.2 Integer and Real Number Objects

Integer- and Real-number objects are similar in concept to the string objects described above. Each object type holds a simple number. The following functions operate on these objects:

dt_alloc_integer() => Handle integer_ob
: Allocates an integer number object. The object's initial value is zero.

`dt_alloc_real() => Handle real_ob`
: Allocates a real-number object. The object's initial value is zero.

`dt_setintegervalue(Handle intob, integer value) => boolean ok`
: Sets the object's value to value.

`dt_setrealvalue(Handle realob, real value) => boolean ok`
: Sets the object's value to value.

`dt_integervalue(Handle intob) => integer i`
: Returns the integer-object's value.

`dt_realvalue(Handle realob) => real r`
: Returns the real-object's value.

5.3 Binary-Data Functions

It is often convenient to use string objects to store "binary" data -- data that are not intended to be printed, and that may contain "NULL" characters and so forth. However, it is also convenient to be able to represent these binary data as printable ASCII characters. For example, Daylight's THOR database system can store arbitrary binary data, but needs to be able to represent it lexically with a restricted set of printable characters.

Three "convenience" functions are provided to convert binary data to printable ASCII and back again. The conversion used maps each 3 bytes of binary data into 4 bytes of ASCII data (i.e. 4 groups of 6 bits are converted to 4 ASCII characters). The ASCII representation has a trailing byte indicating how many of the last 3 binary bytes are part of the original binary data.

Note that these functions take strings, not string objects, as their inputs, and return string objects, not strings.

`dt_ascii2binary(string ascii) => handle string_ob`
: Convert a binary string to its ASCII representation; returns a newly- allocated string object.

`dt_binary2ascii(string binary) => handle string_ob`
: Convert an ascii representation back to its binary form; returns a newly-allocated string object.

`dt_binary2asciilen(string binary) => integer`

Returns the length of the string that dt_binary2ascii() would return, but without allocating any object.

6. Basics: Streams and Sequences

It is often useful to perform some operation iteratively over the constituent parts of an object; for example, one might want to examine the properties of each atom or bond of a molecule. Two special object types, the stream and the sequence, provide a mechanism for doing this conveniently. Conceptually, a *stream* or a *sequence* is an ordered group of objects with a current position in the order.

(Note: To clarify this concept, it is not the same as a set, since one object can appear several times; nor is it like LISP's list, as sequences can't be appended to one another, and there is no concept of the "tail" of a sequence being a valid sequence.)

6.1 Properties

6.1.1 Stream Properties

Streams are used to enumerate the constituent parts of complex objects such as molecules, and are often the only way these constituent parts can be accessed. For example, if m is a molecule object, invoking

```
        dt_stream(m, TYP_ATOM)
```

returns a stream containing all of the atoms in the molecule.

Streams are deliberately limited in their capabilities in order to make them "cheap" (creating a stream of atoms as illustrated above takes very little computing time). In addition to the polymorphic functions that apply to all objects (described in the chapter entitled POLYMORPHIC FUNCTIONS), there are only four operations on streams:

- create a stream
- reset the stream's position to the beginning
- get the next item in a stream
- ask if the stream's position is at its end

Streams usually have a base object -- the object from which they are derived (see dt_base()). Most streams have one base object, and are deallocated if the base object is changed in a way that makes the stream invalid. For example, a stream of atoms from a molecule is deallocated if a new atom or bond is added to the molecule.

Streams have several important properties:

- dt_next(s) always returns objects in the same order. That is, you can step through the stream, reset it, and step through again with the same results.
- Two streams of the same type with the same base object will both return their objects in the same order.
- A copy of a stream behaves identically to the original. This is true even when a copy is made in the middle of an enumeration; in this case dt_next(copy) will continue the enumeration in the middle of the stream exactly as dt_next(orig) will.

- A stream is deallocated (the stream-object is thrown away and its handle revoked) if the base object (the object from which it is derived) is modified. For example, an atom-object stream is deallocated when the molecule containing the atoms is deallocated or structurally modified.

6.1.2Sequence Properties

The properties of a sequence are a superset of stream properties; in addition to those listed above, sequences can perform the following operations:

- ask if the sequence is at its beginning
- insert an object at the current location
- delete the object from the current location
- add an item to the end of the sequence
- go directly to the end of the sequence

6.1.3 Example

The following short code fragment illustrates how one might create both a stream and a sequence. Both the stream and the sequence will contain all the atom-objects from the molecule, but the sequence can later be modified if desired (the function dt_smilin() is documented in a later chapter).

```
#include <dt_smiles.h>
...
char smiles[20];
dt_Handle strm, seq, mol, atom;

/**** create a stream of the atoms in benzene ****/
strcpy(smiles, "c1ccccc1");
mol = dt_smilin(strlen(smiles), smiles);
strm = dt_stream(mol,TYP_ATOM);

/**** copy the atoms from the stream into a sequence ****/
seq = dt_alloc_seq();
while (NULL_OB != (atom = dt_next(strm)))
  dt_append(seq, atom);
...
```

Several other example programs that make use of sequences are supplied in the Daylight "contrib" directory ($DY_ROOT/contrib).

6.2 Functions on Streams and Sequences

`dt_alloc_seq() => sequence`
: Return a new, empty sequence.

`dt_stream(Handle ob, integer typeval) =>stream`
: Returns a stream (an enumeration) of all parts of type `typeval` within the object `ob`.

`dt_next(Handle s) => object`
: Return the next object in the sequence or stream. Return NULL_OB if all items of the stream or sequence have already been returned.

`dt_atend(Handle s) => boolean`
: Returns TRUE if the most recent call to dt_next(s) returned NULL_OB because the end of the stream or sequence was reached. This is useful in cases where a sequence might contain NULL_OB as a valid item.

If dt_next() has not yet been called for the given sequence or stream, or has not been called since the last call to dt_reset() (or any other function that resets the sequence), dt_atend() will return FALSE, even if the sequence or stream is empty. Note that if the most recent call to dt_next() returned something other than NULL_OB, then dt_atend() will necessarily return FALSE.

`dt_reset(Handle s) => boolean`

Resets the sequence or stream so that it begins again with the first item.

6.3 Functions on Sequences Only

The following functions only apply to sequences; they modify a sequence or otherwise perform "direct access" to the objects it contains.

These functions use the concept of a current object. The current object is the one that was most recently returned by dt_next(). When dt_reset() is called, the current object is thought to be an imaginary object before the first actual object; if the dt_next() reaches the end of the sequence, the current object is thought to be an imaginary object after the last object of the sequence.

`dt_append(Handle seq, object ob) => sequence`

Adds ob to the end of seq; ob may be any valid Handle to an object including the value NULL_OB. Return the (modified) sequence. This function also resets the sequence; that is, it has the effect of dt_reset().

Note that, like dt_insert(), it is permissible to add NULL_OB to a sequence; the NULL_OB handle takes a spot like any other handle. When NULL_OB is an item in a sequence, dt_atend() is required to distinguish between the NULL_OB returned when the end-of-sequence is reached and the NULL_OB handles that are part of the sequence.

`dt_atstart(Handle seq) => boolean`

Returns TRUE if the sequence is reset (if the next call to dt_next(seq) would return the first object in seq).

`dt_delete(Handle seq)`

Deletes the current object of the sequence (i.e. the last one returned by dt_next()); make the object that preceded the deleted object be the new current object. dt_next() will return the same value it would have before the deletion occurred.

`dt_insert(Handle seq, object ob) => sequence`

Inserts ob in the sequence before the current object; the newly- inserted object becomes the current object.

The new object is inserted before the object most recently returned by dt_next(), or at the start of the sequence if dt_reset() was the last operation, or at the end of the sequence if dt_atend() would return TRUE.

Note that dt_next() will return the same value it would have before the insertion occurred. A sequence that is reset when an insertion is made is not reset after the insertion, since the newly-inserted object becomes the current object.

It is permissible for `ob` to be NULL_OB; in this case, the NULL_OB handle takes a spot in the sequence. When NULL_OB is an item in a sequence, dt_atend() is required to distinguish the NULL_OB returned when the end-of-sequence is reached from NULL_OB handles that are part of the sequence.

`dt_toend(Handle seq)`

7. SMILES Toolkit: Molecules

A molecule object represents the atoms, bonds, cycles and chiral centers of a molecule. Because it is such a fundamental object in computational chemistry, there are more functions that operate on molecules than any other object. One can:

◊ Produce a molecule from a SMILES string.
◊ Produce a SMILES string or a unique SMILES string from a molecule.
◊ Build a molecule "from scratch" using functions to create an empty molecule, then adding atoms and bonds.
◊ Add and delete atoms and bonds.
◊ Change the properties of atoms and bonds.
◊ Test for aromaticity of a molecule, atom, or bond. Aromaticity is determined automatically for Kekulé structures.
◊ Find symmetry classes for atoms.
◊ Tests for and set chiral features.
◊ Generate streams of the atoms, bond, and cycles of a molecule, and streams of atoms of a cycle, bonds of a cycle, and so forth.

7.1 Creating Molecules

There are two ways to create a molecule object: "From scratch" (allocate an empty molecule), and by parsing a SMILES string:

`dt_alloc_mol() => molecule`
Returns a new, empty molecule.

`dt_smilin(string smiles) => molecule`
Interprets the given SMILES string and return a handle for the resulting molecule structure.

Efficiency Note: The Toolkit's internal representation of molecule objects is designed for efficient analysis of the molecule's properties, and for responding to queries about the molecule quickly. It is not intended to be a compact representation of the molecule, and uses many times more memory to store than a compact representation such as a SMILES string. Applications that require many thousands of molecules in memory simultaneously should use a more compact representation for those molecules that are not of immediate interest.

7.2 Constituents of a Molecule

These functions provide ways to enumerate (generate streams of) the atoms, bonds, cycles, and chiral features of molecules. Also included are two functions, dt_bond() and dt_xatom(), for accessing related constituents without the necessity of creating a stream.

`dt_stream(Handle ob, integer typeval) => stream`
Generate a stream of atoms, bonds or cycles -- a stream that contains all of the objects of the specified type that are part of the object.

`Object` can be a molecule, atom, bond or cycle. For example, a stream of `dt_stream(bond, TYP_ATOM)` returns the two atoms at either end of the bond; a stream of `dt_stream(cycle, TYP_BOND)` returns all the bonds that are part of the cycle.

Note: remember, `dt_stream()` *is polymorphic -- it applies to other objects, too. Here, we are only discussing the molecule and its constituent parts.*

`dt_canstream(Handle object,Integer type, boolean iso, boolean addh) => stream`

Allocates a stream of type 'type', in canonical order, for the molecule or reaction 'object'. `Object` can be a molecule, atom, bond or cycle.

`dt_origstream(Handle object,Integer type) => stream`

Returns a stream of objects in which the objects appear in "original" order. That is, dt_next() will return atoms in the same order as they appear in the original string used to create the object molecule via dt_smilin(), or in the order in which they were added to molecule using dt_addatom().

`dt_bond(Handle at1, Handle at2) => bond`

Returns the handle of the bond joining the two atoms.

`dt_xatom(Handle a, Handle b) => atom`

Return the atom that is across the bond b from the atom a.

`dt_uid(Handle abc) => integer`

Returns the unique id of an atom, bond or cycle within the containing molecule. A unique id is a smallish non-negative integer (i.e. it can be zero) that is guaranteed to not change for as long as the object `abc` exists. The intention is that unique id's, unlike handles, be reasonably dense; for this reason the uid makes a good array index but a handle does not. Note that unlike handles, uid's are only unique across a single containing object; for example, atoms from two different molecules may have the same uid.

`dt_uidrange(Handle molecule, integer typ) => integer`

Returns a number that is at least 1 greater than the largest uid currently associated with any constituent having type `typ` contained in the molecule.

7.3 Modifying Molecules

7.3.1 Derived Properties

Many molecule properties are *derived properties*. Derived properties are not explicitly specified as you create the molecule; rather, they are computed once the molecule is assembled. For example, you don't directly add a cycle (a ring) to a molecule; instead you add various bonds between the molecule's atoms; the Toolkit detects the existence of cycles after a molecule's atoms and bonds are completely specified. Cycles are thus a derived property. Other derived properties include aromaticity, chirality and, in some cases, bond type (see also dt_bondtype() and dt_bondorder()).

7.3.2 The Modify-on and Modify-off States

Before a molecule object can be modified it must be put into the modify-on state; when modifications are complete, the molecule object is returned to the modify-off state. Generally speaking, functions that modify significant properties of a molecule or its constituents may be applied only in the modify-on state. These functions are further divided into structural-modification functions (described below) which change the structure of the molecule, and non- structural-modification functions, which merely change the properties of the existing structure of the molecule.

These modify-on and modify-off states serve two purposes. First, when modifying a molecule or building one "from scratch," the molecule may enter temporary configurations in which it does not represent a valid chemical compound. The modify-on state indicates that the molecule may be in such a state, and prevents the application from asking questions (such as questions about derived properties) that the Toolkit may not be able to answer. Second, some of the derived properties take a significant amount of time to compute (e.g. finding a "smallest set of smallest rings" is a computationally difficult task for which no fast algorithm exists). The transition from modify-on to modify-off tells the Toolkit to recompute derived properties as necessary.

`dt_mod_on(Handle m) => boolean`
: Puts the given molecule into the modify-on state; molecules in this state may be modified.

`dt_mod_off(Handle m) => boolean`
: Puts the given molecule into the modify-off state. This function causes the molecule's structure to be analyzed; its properties may be changed as a result. The most notable change is to the aromaticities of constituents (atoms, bonds, and cycles). A recalculation of contained cycles may also take place.

  If there is an error, the molecule is deallocated just as though dt_dealloc() had been called. (This is an unfortunate side-effect of the structure-analysis functions: if they fail, they leave the molecule in an unusable state. Molecules that are "precious" should be copied just prior to invoking dt_mod_off(); if it returns TRUE the copy can be discarded. The copy-and-discard operation is "cheap" (i.e. fast) compared to the structural analysis.)

`dt_mod_is_on(Handle m) => boolean`
: Returns TRUE if the molecule is in the modify-on state, FALSE otherwise.

7.3.3 Functions Applicable Only During Modify-On

These functions can only be applied to a molecule or its constituent parts when the molecule is in the modify-on state. Generally speaking, such functions modify the structure of a molecule in some significant way.

```
dt_addatom()
dt_addbond()
dt_dealloc() (when applied to an atom or bond)
dt_setbondorder()
dt_setbondtype()
dt_setcharge()
dt_setchival()
dt_setdbo()
dt_setimp_hcount()
dt_setnumber()
dt_setweight()
```

7.3.4 Functions Applicable Only During Modify-Off

These functions can only be applied to a molecule or its constituent parts when the molecule is in the modify-off state. Generally speaking, such functions only make sense when applied to well-formed molecules.

```
dt_arbsmiles()
dt_cansmiles()
dt_symclass()
dt_symorder()
dt_xsmiles()
```

7.3.5 Functions Applicable At All Times

All functions not listed either here or in the previous section that normally apply to molecules can be applied to a molecule in both the modify-on or the modify-off states.

7.4 Structural-Modification Functions

The three functions dt_addatom(), dt_addbond(), and dt_dealloc() (when applied to atoms or bonds) are collectively referred to as *structural modification functions*. After calling a structural modification function, future streams returned by dt_stream() are no longer guaranteed to return objects in the same order that they were returned before the modification. Note that this remains true even if the structure of the molecule is later restored to an equivalent form.

Also, remember that any structural modification to a molecule causes all streams of atoms, bonds or cycles over the molecule to be deallocated.

`dt_addatom(molecule m, integer atno, integer hcount) => atom`
: Add an atom with atomic number `atno` and `hcount` hydrogens to the given molecule.

`dt_addbond(atom a1, atom a2, integer btype) => bond`
: Add a bond with the given bond type between the two atoms.

`dt_dealloc(object ab) => boolean`
: Atoms and bonds are removed from a molecule by deallocating them.

7.5 Properties of Atoms

Arbitrary SMILES: An *Arbitrary SMILES* is derived by the same algorithm as a *unique SMILES*, except that a user-specified set of labelings is used, allowing the generation of a SMILES in an arbitrary order. The user-specified labeling of each atom is called the *arbitrary order* of the atom. The SMILES begins with the atom whose arbitrary order is lowest; when branch points are reached, the branch with the atom whose arbitrary order is lowest is written first. The following functions are related to *Arbitrary SMILES*:

`dt_setarborder(atom at, integer order) => boolean`
: Sets the atom's arbitrary order value

`dt_arborder(atom at) => integer`
: Returns arbitrary order value for the given atom.

`dt_arbsmiles(molecule m, boolean iso) => string`
: Returns an Arbitrary SMILES string for the given molecule. The `iso` parameter indicates whether the returned SMILES string should contain isomeric labelings.

Atomic Charge: Two functions are provided to set and get the charge on an atom:

`dt_setcharge(atom at, integer charge) => boolean`
: Sets the atom's formal charge.

`dt_charge(atom at) => integer`
: Returns the atom's formal charge.

Hydrogen Count: The graphs used to represent molecules are usually hydrogen- suppressed: hydrogens are represented as a property of the "heavy" atoms to which they are attached rather than as separate atom objects. Such hydrogens are called *implicit hydrogens*. In some cases hydrogens must be actual objects (e.g. when there is isotopic information or more than one bond to the hydrogen); in other cases it may be convenient to have hydrogen objects (e.g. when data, such as xyz coordinates, are known about them). Such hydrogens are called *explicit hydrogens*.

The following functions are used for implicit and explicit hydrogens (also see dt_addatom()):

`dt_hcount(atom at) => integer`
: Returns the total number of hydrogen atoms (implicit and explicit hydrogens) bonded to the atom.

`dt_imp_hcount(atom at) => integer`

Returns the number of implicit hydrogens bonded to the atom.

`dt_setimp_hcount(atom at, integer count) => boolean ok`

Sets the number of implicit hydrogens on the atom.

Atomic Number, Symbol, and Weight: An atom's atomic number and weight are independent in the Daylight Toolkit. In real life, only certain isotopes exist for each atomic number; the Daylight Toolkit imposes no such constraint.

The atomic symbol is derived directly from the atomic number; the Toolkit doesn't provide a way to set it independently.

`dt_number(atom at) => integer`

Returns the atom's atomic number.

`dt_setnumber(atom at, integer num) => boolean`

Sets the atom's atomic number.

`dt_symbol(atom at) => string`

Returns the atom's atomic symbol (e.g. "C", "Si&quot).

`dt_weight(atom at) => integer`

Returns the atom's atomic weight. The returned weight is '0' if the weight is unspecified (eg. the default weight of an atom). Returns an integer weight for atoms which have been set to a specific isotope value with dt_setweight().

`dt_setweight(atom at, integer weight) => boolean`

Sets the atom's atomic weight.

7.6 Properties of Bonds

Bond type and *bond order* are closely related but not identical properties of a bond object.

Bond order: a formal property of the bond, which can only be one of DX_BTY_SINGLE, DX_BTY_DOUBLE, DX_BTY_TRIPLE, representing single bonds, double bonds, and triple bonds, respectively.

Bond type: a *derived property*, which is normally computed by the Toolkit when the molecule goes from modify-on to modify-off. The primary situation where the bond type will differ from bond order is in aromatic structures, in which single and double bonds can be converted to aromatic bonds. Bond type can be any of DX_BTY_SINGLE, DX_BTY_DOUBLE, DX_BTY_TRIPLE, or DX_BTY_AROMAT.

When a molecule is in the modify-on state, a bond's type or order can be changed. Normally, one specifies a bond's type, and lets the Toolkit generate the bond order from that. If you specify a bond's order via dt_setbondorder(), its type will be changed too. If you specify its type via dt_setbondtype(), the bond order may not agree until dt_mod_off() is called.

`dt_bondtype(Handle bond) => integer`

Returns the bond's type. This value can change when a molecule changes from the modify-on state to the modify-off state (For example, it might change from single or double to aromatic).

`dt_setbondtype(Handle bond, integer type) => boolean ok`

Sets the bond's type. Also may affect the bond's order; if the bond type is set to single, double, or triple, the bond order is too; if the bond type is set to aromatic, bond order becomes unknown.

`dt_bondorder(Handle bond) => integer order`

Returns the bond's order.

`dt_setbondorder(Handle bond, integer order) => boolean ok`

Sets the bond's order. Also affects the bond's type, which is also set to the value `order`.

7.7 Properties of Cycles

There are no specific functions for accessing or modifying cycles in a molecule, as cycles are a derived property of the bonds. The general function dt_stream() will return the cycles of a molecule, bond, or atom.

7.8 Generating SMILES

`dt_cansmiles(molecule m, boolean iso) => string`

Returns a canonical SMILES string for the given molecule. (Note that this causes calculation of the canonical labelings if it has not yet been done, a potentially time-consuming operation.) The `iso` parameter tells whether the SMILES string should contain isomeric labellings (isotopic and chiral information). (A canonical SMILES string with isomeric labelings is called an *Absolute SMILES*. Without isomeric labelings, it is called a *Unique SMILES*.). The molecule must be in the modify-off state (see dt_mod_off()).

Note: The string returned is part of the molecule object and may change or be discarded if the molecule is modified or deallocated. In general, you should copy the string if you will need it later.

`dt_xsmiles(molecule m, boolean iso, boolean explicit) => string`

Returns an exchange SMILES string for the given molecule. An exchange SMILES is a SMILES with Daylight aromaticity conventions eliminated. The `iso` parameter tells whether the SMILES string should contain isomeric labellings (isotopic and chiral information). The explicit parameter tells whether to also explicitly list attached hydrogens for all atoms. The molecule must be in the modify-off state (see dt_mod_off()).

Note: The string returned is part of the molecule object and may change or be discarded if the molecule is modified or deallocated. In general, you should copy the string if you will need it later.

7.9 Aromaticity

These functions test the aromaticity of molecules, atoms, bonds and cycles, and where appropriate, allow you to set those attributes.

Aromaticity in the Daylight Toolkit is a complex subject. For a more thorough discussion of aromaticity in the Daylight System, please see SMILES Chapter of the Daylight Theory Manual.

`dt_aromatic(object ob) => boolean`

Returns TRUE if the given object (an atom, bond, cycle or molecule) is considered aromatic.

`dt_setaromatic(atom at, boolean isarom)`

Sets the aromaticity of the atom at to TRUE or FALSE according to the value of isarom.

7.10 Symmetry

The Daylight Toolkit can compute the symmetry of a molecule. There are two different symmetry values you can access.

Symmetry Class: Two atoms in a molecule will be in the same *symmetry class* if and only if they are symmetrically equivalent. The actual number assigned to a symmetry class is arbitrarily -- the the only significance of the numbers is whether two atoms have the same class number or not.

Symmetry Order: The algorithm that generates the symmetry order uses graph invarients (including the *symmetry classes* described above) to generate a unique labeling (the *symmetry order*) of the molecule's graph. An atom's symmetry order controls the generation of the *Unique SMILES* (see dt_cansmiles()).

Note that any change, however slight, to the molecule may cause the symmetry class and/or symmetry order values to change.

`dt_symclass(atom at) => integer`
: Returns the unique symmetry class of an atom in its parent molecule.

`dt_symorder(atom at) => integer`
: Returns the unique symmetry order of an atom in its parent molecule.

7.11 Chirality

The most complex attributes are chirality attributes, which are specified by single integer codes called chiral values. These values combine two separate pieces of information, a chiral class (corresponding to a geometric configuration such as tetrahedral, octahedral, and so on) and a chiral order (a particular ordering around the chiral center, such as clockwise, counter-clockwise, and so on).

Symbolic constants are defined to simplify the specification of chiral values. In the current implementation, only cis/trans and tetrahedral chirality are supported. The following symbolic constants combine the chiral class and chiral order information for convenience:

Cis/Trans Chirality	
DX_CHI_NO_DBO	cis/trans situation, but chirality is unspecified
DX_CHI_CIS	cis configuration around a double bond
DX_CHI_TRANS	trans configuration around a double bond
Tetrahedral Chirality	
DX_CHI_NONE	unspecified chirality
DX_CHI_THCCW	tetrahedral center with counterclockwise configuration
DX_CHI_THCW	tetrahedral center with clockwise configuration

`dt_dbo(bond db, bond b1, bond b2) => integer`
: Returns the "double-bond orientation" between `b1` and `b2`. The bond `db` should be a double bond that is at the center of a cis/trans configuration. Bonds `b1` and `b2` should single bonds attached to the atoms at the end of `db`, one on each of the two atoms. The return value will be equal to one of the symbolic constants DX_CHI_CIS, DX_CHI_TRANS, or DX_CHI_NO_DBO. The latter case indicates that the cis/trans configuration around db is unspecified.

`dt_setdbo(bond db, bond b1, bond b2, integer dboval) => boolean`
: Sets the "double-bind orientation" between b1 and b2 to the given value. The first three parameters are as described above for dt_dbo(). The last parameter is one of DX_CHI_CIS, DX_CHI_TRANS, or DX_CHI_NO_DBO.

`dt_chival(atom at, sequence seq) => integer`
: Returns the chiral value around the given chiral center `at`, determined with respect to the order of the bonds in this sequence. See the function's full description for details.

`dt_chiseq(atom at, integer chival) => sequence`

Returns a sequence of bonds having the chirality given by chival around the given atom at (the chiral center). the chiral order portion of the value is used to determine the ordering of the returned sequence. See the function's full description for details.

`dt_setchival(atom at, sequence seq, integer chival) => boolean`

Sets the chiral value at the given chiral center at. The parameter seq is a sequence of bonds that meets the conditions specified for dt_chival(); the chiral value is set with respect to the order of bonds in this sequence.

`dt_chiperm(sequence seq, bond start, integer chival) => sequence`

Given a sequence of bonds having the given chiral value, modify it (i.e., permute it) so that the chiral value is preserved, but so that it begins with the given bond `start`.

`dt_chiclass(integer chival) => integer`

Return an integer code for just the chiral class part of the given chiral value.

`dt_chiorder(integer chival) => integer`

Returns an integer code for just the chiral order portion of the given chiral value.

`dt_isohydro() => atom`

Returns a hydrogen-atom object that is useful for representing implicit-hydrogen atoms in calls to the isomeric functions. Each call to this function returns the same special atom. The atom may not be modified (attempts will fail) and it has no parent molecule (calls to dt_parent() will return NULL_OB). In general, applications should not attempt to play around with it too much; its only intended use is in calls to the isomeric functions defined above.

8. SMILES Toolkits: Substructures and Paths

8.1 Introduction

The *Daylight Substructure Toolkit* provides objects and functions to represent and operate on substructures and paths:

Substructure:

A set of atoms and bonds from one molecule. Typically a substructure is obtained as the result of a substructure search (see the chapter on the SMARTS Toolkit), but they can be created "from scratch" by an algorithm of your own design using Toolkit functions, described below.

A substructure is simply a set -- there is no implied order to the atoms or bonds in the set, and each atom and bond from the molecule occurs at most one time in the substructure. We normally think of a substructure as a set of atoms and bonds that are connected together in some chemically meaningful way. A substructure object can be used to represent these "ordinary" substructures, but it can also be used to represent less conventional collections. For example, a substructure object could hold all of the double bonds in a structure, all of the atoms with an odd number of protons, and so forth. In other words, a substructure object is just an arbitrary set of atoms and bonds; it is up to the programmer using the object to decide what the set means.

Path:

A path through a substructure. That is, a set of atoms and bonds from a single base molecule, and a particular ordering of those atoms and bonds.

The word "path" suggests that the ordering in the object be related to the actual connectivity of the molecule (as though you could "walk" the path without jumping around), but this is not a requirement. The path object is only defined to be an ordered set of atoms and bonds. For example, a path object could contain all bonds ordered by their bond order (i.e. single, double, triple, then aromatic), or could contain all atoms ordered by atomic number. Like the substructure object, it is up to the programmer to assign meaning to the path object's contents.

The SMARTS Toolkit also uses a closely-releated object type, the pathset:

Pathset:
A set of zero or more path objects from a single molecule. The pathset object and its uses are discussed at length in the SMARTS Toolkit chapter .

NOTE: *An often confusing point is that the SMILES Toolkit provides substructure and path objects, but does not do substructure searching (substructure searching is available in the SMARTS Toolkit -- sold separately). There are many sources of substructures and paths besides SMARTS; the path and substructure objects provide a convenient way to represent them whether or not you purchase the SMARTS Toolkit.*

To retrieve the contents of a path or substructure, you can create streams of atoms or bonds (see dt_stream()). Any modificacation to a path or substructure (adding or removing an atom or bond) causes all streams to be deallocated.

Substructure and Path objects always have a molecule as their *base object* (see dt_base()). Their existance dependes on the existance of the base molecule; deallocating a molecule causes all paths and substructure objects of the molecule to be deallocated.

8.2 Functions on Substructures and Paths

`dt_alloc_substruct(Handle mol) => substruct`
Returns a new substructure object. The new substructure object initially is empty (contains no atoms or bonds).

`dt_alloc_path(Handle mol) => path`
Returns a new path object. The new path object initially is empty (contains no atoms or bonds).

`dt_add(Handle sp, Handle ab) => boolean`
Add an atom or bond to the substructure or path. The atom or bond must be from the same molecule as the substructure's or path's base object

Adding an object to a substructure simply adds it if it is not there; the order in which objects are added to a substructure is not remembered. Adding an object to a path adds it to the end of the path unless it is already in the path, in which case the requested addition is ignored.

`dt_member(Handle sp, Handle ab) => boolean`
Returns TRUE if and only if the given atom or bond is a member of the substructure or path.

`dt_remove(Handle sp, Handle ab) => boolean`
Remove an atom or bond from a substructure or path.

Removing an atom or bond from a substructure may cause its ordering to change in arbitrary ways. Removing an atom or bond from a path leaves the order of the remaining objects unchanged.

9. SMARTS Toolkit: Structural Searching

9.1 Introduction

The Daylight SMARTS Toolkit provides a powerful set of substructure searching algorithms. The SMARTS Toolkit can parse a SMARTS string and produce a pattern object, then test the pattern against one or more molecule objects to see if the molecule contains the pattern. Several types of pattern searches are available, including "yes/no" tests and exhaustive enumeration of all occurances of a pattern in a molecule.

There are three objects specific to the Daylight SMARTS Toolkit:

pattern
: The result of "compiling" a SMARTS string. Each SMARTS string is converted to a pattern before use; the pattern-generation algorithm parses the SMARTS, checks for errors, and pre-computes certain information that improves substructure-search speed.

pathset
: A collection of path objects all of which have the same base molecule. A pathset is the result of matching a pattern against a molecule. (For background information, see the chapter on Substructures and Paths.)

vbind
: A "vector binding". A binding of a name to a pattern or pathset. This is explained in more detail below.

9.2 Optimizing SMARTS

`dt_smarts_opt(string smarts, boolean vmatch) => string optsmarts`

Returns a new SMARTS string that is equivalent in meaning to the given SMARTS, but that has been reordered to permit matches to be done more quickly on typical molecules. (See the chapter on SMARTS in the section entitled "Efficiency Considerations" for more information.)

The optimized SMARTS is called "equivalent in meaning" to the original because the optimized string will match the exact same sets of objects as the original (though the pathsets returned may have their atoms and bonds in different orders), no matter what molecules they are matched against.

The exact meaning of a SMARTS depends upon the type of match to be performed, as described below in the functions dt_match(), and dt_vmatch(). The parameter `vmatch`, if TRUE, indicates that the SMARTS string should be optimized for use by dt_vmatch(); otherwise indicates that it should be optimized for dt_match() or dt_umatch(). (In practical terms, optimizing for dt_match() or dt_umatch() means that the head atom of the optimized string may not be the same as that of the original string, whereas optimizing for dt_vmatch() means that the first atom of the optimized string will be the same as that of the original string.)

The optimized SMARTS returned by this function are expected to be matched reasonably quickly on the average. The actual matching speed, however, depends on the molecules to be matched against, as well as on the SMARTS string itself. It is not possible to guarantee that the returned string will actually match faster than to original SMARTS. It is often possible to generate faster SMARTS "by hand" using particular knowledge of the molecules to which it will be applied. The algorithm employed by this function is based on rules generated from a database of "typical" organic molecules.

9.3 Allocating Patterns and Pathsets

`dt_smartin(string smarts) => Handle pattern`
: Interprets the SMARTS string and creates a pattern object; returns the object's handle.

`dt_alloc_pathset(Handle molecule) => Handle pathset`
: Allocates a new pathset object and returns its handle. (Note: this function is typically not needed; pathsets are normally generated by dt_match(), described below.

9.4 Vector Bindings and Vbind Functions

Vector bindings are a mechanism that allow faster evaluation of a match, and allows complex patterns to be constructed out of simpler patterns. Those familiar with earlier Daylight software (versions 3.6x and earlier) will recognize vector bindings as closely related to GCL's "define" functionality.

The term "binding" comes from mathematics. When one thing is bound to another, the latter can be used to represent the former. For example, in high-school algebra, the expression "x=3" binds the value three to the symbolic name "x"; thereafter, wherever "x" appears we substitute 3.

9.4.1 Pattern Bindings

The simplest use of vector bindings is when a pattern is bound to a name. Once the pattern and name are bound, the name can be used in another SMARTS string in place of an atomic symbol. For example, if we bind the pattern for C[Cl,Br,I] to the name "HALO", the string [$HALO] becomes a valid atomic symbol; wherever it is used it is equivalent to the atomic expression [$(C[Cl,Br,I])]. (This is an example of a *recursive SMARTS*.) Binding patterns to names doesn't extend the expressive power of SMARTS (unnamed vectors such as [$(C[Cl,Br,I])] are valid in any SMARTS), but it tends to make SMARTS more readable. This is especially true where atomic expressions such as [$(C[Cl,Br,I])] occur multiple times. For example, a 1,2,4-substituted cyclohexane could be written two ways; the second is easier to write and easier to understand:

```
[$(C[Cl,Br,I])]1[$(C[Cl,Br,I])][CH2][$(C[Cl,Br,I])][CH2][CH2]1

[$HALO]1[$HALO][CH2][$HALO][CH2][CH2]1
```

9.4.2 Pathset Bindings

When a pathset is bound to a name, that name represents all the places in the molecule where a search has already succeeded. If, for example, one has a number of SMARTS to test against a single molecule, and many of the SMARTS contain an common expression such as [$HALO], the search speed of each can be improved by pre-computing the pathsets for [$HALO] (i.e. doing dt_vmatch() on the pattern for C[Cl,Br,I]), then binding the resulting pathset to the name "HALO". When the [$HALO] atomic expression is encountered in a SMARTS, the vector binding already contains the pathset with all atoms that match; no additional searching is required for that atomic expression.

Binding pathsets in this fashion can greatly improve the performance of "chemical knowledge" systems in which a large number of rules are applied to each molecule. By carefully choosing common subexpressions, matching them to the molecule, and binding the resulting pathsets to names, a great deal of redundant searching can be eliminated.

9.4.3 Functions

`dt_alloc_vbind(string name) => Handle vbind`

If there is already a vector binding with the specified name, that existing vector binding is returned. Otherwise, a new vector binding is allocated and given the specified name.

The string name must begin with a letter from the alphabet; subsequent characters in the name must be alphabetic, a digit, or the underscore '_'. In UNIX parlance, the name must match the regular expression `/^[a-zA-Z][a-zA-Z0-9_]*$/`.

If the object bound to a vector-binding is deallocated (see dt_setval(), below), the vector binding's value is automatically set to NULL_OB. (Note that the vector-binding object is not deallocated in this case since its binding is not its parent or base object.)

`dt_name(Handle vbind) => string name`

Returns the vector binding's name.

`dt_setval(Handle vbind, Handle pp) => boolean ok`

Sets the value of the vector binding to the value of `pp`, which must be NULL_OB, a pattern, or a pathset.

`dt_getval(Handle vbind) => Handle pp`

Return the value of the vector binding vbind. It will be NULL_OB, a pattern, or a pathset.

9.5 Pattern Matching

All of the matching functions require the target to be a valid, consistant object. This means that molecules and reactions must be in mod-off state before calling any of the following functions.

`dt_match(Handle pat, Handle mol, boolean exist) => pathset`

Matches the pattern against the molecule; returns a pathset indicating the results of the match, or NULL_OB if no match is found or an error is detected.

The boolean parameter `exist` indicates whether an exhaustive search or a first-only search is to be performed. If exist is FALSE, an exhaustive search takes place; pathset will contain *all* places in the molecule where pattern matches. If `exist` is TRUE, the match stops as soon as the first match is found; pathset will contain just the first path that an exhaustive search would have found.

`dt_vmatch(Handle pat, Handle mol, boolean exist) => pathset`

Matches the pattern pat against the molecule mol as with dt_match(), above, except that each of the paths in the returned pathset contains just a single atom, the one that matched the "head" atom of the SMARTS from which pat is derived. Further, no two paths in pathset will contain the same atom.

`dt_umatch(Handle pat, Handle mol, boolean exist) => pathset`

A *unique set of atoms* match. Matches the given pattern against the molecule as with dt_match(), above, except that each of the paths in the returned pathset is guaranteed to contain a unique set of atoms (relative to all other paths in the pathset).

Conceptually one can think of this as though dt_match() were performed, then all paths compared to one another. If two paths contain the same atoms, one of the two (the choice is arbitrary) is removed from the pathset. Notice that two paths thus compared may not be equivalent; in particular paths that include cycles may contain the same atoms but not the same bonds. The resulting pathset is guaranteed to contain (as determined by dt_member()) the exact same number of atoms as a match performed with dt_match(), but it may not contain the same number of bonds.

The purpose of this function is to eliminate the "uninteresting" redundancy in paths that are typically returned by exhaustive searches using dt_match(). An exhaustive search, by definition, will find a path for each symmetry in the pattern. For example, dt_match() will return 12 paths when the SMARTS c1ccccc1 is applied to the molecule c1ccccc1; once for each of 6 possible starting atoms, times two for the clockwise and anticlockwise paths. When dt_umatch() is applied, only one path is returned.

The parameter exist is included for consistency; note, however, that if exist is TRUE, dt_match() and dt_umatch() are equivalent.

`dt_xmatch(Handle pat, Handle mol, integer limit) => pathset`

This is a further restriction of the *unique set of atoms* match used by dt_umatch(3). In this case, only non-overlapping paths are returned. That is, no two paths in the result set will contain the same atom. The parameter 'limit' works as follows: if 'limit' is zero, returns all possible paths (exhaustive search). If 'limit' is a positive integer, returns up to 'limit' answers. Hence, 'limit' can be used to restrict the number of matches found.

NOTE: the results from dt_xmatch(3) depend on the processing order of the atoms in the target molecule or reaction. For example, matching the pattern 'CC' against the molecule 'CCCC' will return two pathsets, Matching the same pattern against 'C(CC)C' gives one pathset. In the first case, Carbons #1 and #2 are matched first, leaving #3 and #4 as the second match. In the second case, Carbons #2 and #3 are matched first, leaving #1 and #4, which don't match further.

Logically, dt_xmatch(3) performs an exhaustive match and then picks one of potentially multiple non-overlapping sets of paths for the answer. Fortunately, although one can think of dt_xmatch(3) working this way, the actual implementation is much faster.

10. Fingerprint Toolkit

10.1 Introduction

Fingerprints, their uses and the history of their development in the Daylight Toolkit (tm) are described in detail in the Daylight Theory Manual, chapter on Fingerprints.

The Daylight Fingerprint Toolkit provides a set of tools for rapidly screening very large databases of chemical structures for substructure searching, and for computing the structural similarity between molecules.

Those who have seen or used Daylight's Merlin program will immediately recognize that the Fingerprint Toolkit is part of the foundation of Merlin. However, it should be noted that Merlin has many more capabilities than just the functionality available via the Fingerprint Toolkit; fingerprinting is only a base on which a much larger set of capabilities is built.

The Fingerprint Toolkit, unlike other Daylight Toolkits, *is not recommended for most programming projects*. It is intended for a few special situations where customers have existing database-searching capabilities and wish to enhance performance or add similarity metrics. If you are contemplating building a chemical information system, we strongly recommend that you consider using Merlin and THOR rather than starting with the Fingerprint Toolkit.

10.2 Fingerprint Functions

The Daylight Fingerprint Toolkit uses Fingerprint Objects to represent fingerprints. Fingerprints have the following properties:

Fingerprint Properties	
bitmap	the fingerprint itself
number of bits	the number of bits in the fingerprint (its length in bits)
orig number of bits	the number of bits in the original fingerprint (before folding)
number of bits set	the number of 1's in the fingerprint's bitmap
orig num. bits set	the number of 1's in the original fingerprint's bitmap (before folding)
version	the version of the Daylight Toolkit used to create the fingerprint

10.2.1 Global Settings

In versions prior to 4.42, there were three global toolkit values which controlled fingerprint creation size and folding. These are no longer needed.

10.2.2 Creating Fingerprints

There are two functions to create fingerprints. You can allocate a "blank" fingerprint, then fill it in later with data from an external source (see Fingerprint Bit Operations, below), or you can create a fingerprint directly from a molecule.

`dt_fp_allocfp(integer size) => Handle fingerprint`

Allocate an empty fingerprint. The fingerprint's size will be the given "size" value. It is an error to apply any function that returns a property of the fingerprint unless that property has been explicitly set.

`dt_fp_generatefp(Handle ob, integer minstep, integer maxstep, integer size) => Handle fingerprint`

Allocate a fingerprint object of the given size; fill its fields with a fingerprint generated from the object ob, then set the objects "original size", "original bits set", "size" and "bits set" properties (see dt_fp_obitcount(), dt_fp_obits(), dt_fp_bitcount() and dt_fp_nbits()).

The object ob can be any object for which dt_stream(ob, TYP_ATOM) and dt_stream(ob, TYP_BOND) will return a stream of atoms and bonds, respectively. Typically ob is a molecule object, but one can fingerprint various substructures using path, pathset, substructure, cycle, atom, or bond objects. For example, one can produce a "ring- system fingerprint" using a substructure object that contains all of the atoms and bonds in all cycles of a molecule.

The parameters "minstep" and "maxstep" control the fingerprint generation. "minstep" sets the minimum-length path to be included in the fingerprint; "maxstep" sets the maximum-length path included.

`dt_fp_partfp(Handle part, Handle ob, integer minstep, integer maxstep, integer size) => Handle fingerprint`

Like dt_fp_generatefp(3), except only sets the fingerprint for paths which include the object 'part', which may be an atom or bond. This function performs the full path enumeration over 'ob', but only sets bits in the resulting fingerprint for paths containing 'part'.

This function is a supported version of the function previously included in the contrib/stigmata directory. Note that the results using this function will be slightly different, because this version correctly includes branch and cycle paths containing the object 'part'. The contributed version only considered the straight-chain paths containing 'part'.

10.2.3 Properties

`dt_fp_nbits(Handle fp) => integer nbits`

Return the fingerprint's size (number of bits in the bitmap).

`dt_fp_obits(Handle fp) => integer obits`

Return the fingerprint's original size (before folding). This is the value of "size" which was provided when the fingerprint was created (see above).

`dt_fp_bitcount(Handle fp) => integer bitcount`

Return the number of 1's (bits set) in the fingerprint's bitmap.

`dt_fp_obitcount(Handle fp) => integer obitcount`

Return the original bitcount (before folding).

`dt_fp_setobitcount(Handle fp, integer obc) => boolean ok`

Set the original bitcount. This is intended to be used only with fingerprint objects created via dt_fp_allocfp() and filled manually.

`dt_fp_setobits(Handle fp, integer ob) => boolean ok`

Set the original number of bits. As with dt_fp_setobitcount(), only for use with fingerprint objects created via dt_fp_allocfp().

10.2.4 Fingerprint Bit Operations

These operations allow the user to manipulate the individual bit-values of the fingerprint. They are useful for creation of custom fingerprints (eg. bitscreens, 3-D or spectral fingerprints), combining multiple fingerprints, or writing specialized comparison functions.

Note that dt_stringvalue() and dt_setstringvalue() can be used to get and set the entire binary value of a fingerprint. The functions described here are useful for manipulating individual or ranges of bits within a fingerprint.

`dt_fp_bitvalue(Handle fp, integer bitno) => integer value`

Returns the current value of a bit in the fingerprint.

`dt_fp_setbitvalue(Handle fp, integer bitno, integer value) => boolean ok`

Sets the current value of a bit in the fingerprint to "value".

`dt_fp_range(Handle fp, integer offset, integer nbits, integer *soffset) => string range`

Gets a range of bits from a fingerprint. The range is returned as a string of binary data, starting "soffset" bits from the beginning of the string. The range of bits requested begins at "offset", for "nbits" bits, or to the end of the fingerprint.

`dt_fp_setrange(Handle fp, integer offset, integer nbits, integer slen, string string, integer soffset, integer operation) => boolean ok`

Sets the values of a range of bits in the fingerprint. The range of bits set are given by "offset" for "nbits" bits. The "operation" and the given string, starting "soffset" bits from the beginning of the string, controls how the bits are set. Legal operations are:

DX_FP_SET	sets each bit in the range to the source (string) value.
DX_FP_NOT	sets each bit in the range to the inverse of source (string) value.

DX_FP_OR	sets each bit in the range to the logical-"or" of the source (string) value and current bit value.
DX_FP_AND	sets each bit in the range to the logical-"and" of the source (string) value and current bit value.
DX_FP_XOR	sets each bit in the range to the logical-"xor" of the source (string) value and current bit value.

10.2.5 Comparisons

`dt_fp_fingertest(Handle patfp, Handle molfp) => boolean sub`

Returns TRUE if all of the bits that are set (1) in `patfp` are also set in molfp; that is, returns the value of the logical expression

```
patfp == (patfp AND molfp).
```

In other words, return TRUE if the molecule that generated `patfp` could be a substructure of the molecule that generated `molfp`.

Returns FALSE if any set bit in `patfp` is not also set in `molfp`, or if the two fingerprints are not compatible (e.g. different sizes), or if either object is not a fingerprint.

`dt_fp_euclid(Handle fp1, Handle fp2) => float dist`

Returns the euclidian distance between fp1 and fp2, or -1.0 if an error is detected (i.e. the fingerprints are not compatible or are not fingerprint objects).

`dt_fp_tanimoto(Handle fp1, fp2) => float tan_coeff`

Returns the Tanamoto coefficient between `fp1` and `fp2`, or -1.0 if an error is detected (i.e. the fingerprints are not compatible or are not fingerprint objects).

`dt_fp_foldfp(Handle fp, integer minsize, float mindensity) => boolean ok`

Fold the fingerprint. Returns TRUE if no errors are detected. Note that folding may not actually occur. Folds zero or more times until the "minsize" or "mindensity" values are reached.

11. Depict Toolkit

11.1 Introduction

Depiction and conformation objects contain sets of Cartesian coordinates for 2-dimensional and 3-dimensional positions (respectively) of atoms in a molecule.

On the surface, depictions and conformations seem similar: one contains (x,y) pairs, the other (x,y,z) triplets. However, there are significant differences between the two:

◊ A *depiction* is a schematic representation of a molecule. It is a visualization tool, but doesn't contain information about the molecule itself.
◊ A *conformation* contains measured or computed data about a molecule. Although it can be used for visualization, it can also be used to compute chemical properties, to uniquely identify a particular physical state a molecule might exhibit, and so forth. Conformations are even used as identifiers - see the Daylight Theory Manual for a complete discussion.

In spite of the fundamental difference in the information contained in these two objects, their representations are similar, so many of the functions that operate one also operate on the other.

There are two key differences in the way the Daylight Toolkit treats conformations and depictions:

1. A 2D depiction can be filled in directly from a molecule using a built-in algorithm that generates x,y coordinates for each atom. No such algorithm is available for 3D conformations. (Such algorithms are available from Daylight as separate programs. They are far more complex and computationally expensive than 2D depiction).
2. The many and varied uses of 3D conformations require flexibility and simplicity. The Daylight Toolkit provides access functions such that applications can perform their own custom operations on conformations. There are no facilities for displaying 3D conformations directly, hence no facilities for adding labels, setting graphics attributes, and so forth. However, functions are provided to rotate a conformation to an arbitrary orientation, and to transform a 3D conformation to a 2D depiction by projecting it on to the XY plane.

Although the units used for depictions and conformations can be considered arbitrary, the Toolkit assumes that they are in units of angstroms for the purposes of drawing atom labels in a depiction. If a depiction is made in which the bonds are longer, labels will be correspondingly smaller, and vice-versa.

11.2 Depictions

A depiction is a two-dimensional representation of a molecule. As such, it contains (x, y) coordinates for the the atoms in the molecule, and several attributes that control the graphical appearance of the depiction.

The Daylight Toolkit does not have any I/O capabilities built in. Instead, it relies on a Drawing Library consisting of about a dozen functions. Although several versions of the Drawing Library are available from Daylight, including one for serial graphics terminals and one for the X Window System, the Drawing Library is not considered part of the Toolkit per se: It is possible to write your own Drawing Library and attach it to the Toolkit. The specifications required by the Daylight Toolkit of the Drawing Library are given later in this chapter.

Each graphical attribute of a depiction is represented in the Toolkit by a unique integer value called a GA (**G**raphical **A**ttribute). Each GA represents a complete specification of all graphical features such as color, line width, and line style. It is up to the Drawing Library and the application program to "agree" on the meaning of each GA; the Daylight Toolkit does not know or care about the meaning of a GA; the Toolkit associates GAs with objects (atoms, bonds, etc) and passes them along to the Drawing Library when a depiction is drawn.

The Drawing Library will always accept one special graphics attribute, named DX_GA_UNSPEC. It represents an unspecified graphics attribute; reasonable defaults are used for objects whose graphics attributes are unspecified. The following functions are used for depiction objects:

`dt_alloc_depiction(molecule molec) => depiction`

> Creates and returns a depiction for the given molecule. Note that there is no theoretical limit to the number of pictorial representations that may be created for the same molecule, though an implementation may impose a practical limit. The initial properties of the depiction are as follows: All initial (x, y) coordinate values are 0; the isomeric, compressed, and schematic attributes are all initially FALSE.
>
> Each depiction object has a base object: the molecule it represents (see dt_base()). If the base molecule is modified (i.e. if dt_mod_on() is invoked), the depiction will automatically be deallocated and its handle revoked.

`dt_depict(depiction pic) => boolean`
: Draws a two-dimensional rendition of the underlying molecule using calls to the Drawing Library. The drawing uses the current coordinates and attributes of the given depiction.

`dt_calcxy(depiction pic) => boolean`
: Sets the coordinates for the atoms of the given depiction (i.e. generates a 2D schematic representation), using an internal algorithm. Note that the coordinates are set according to the current attributes of the depiction (i.e. whether it is to be drawn in isomeric, compressed or schematic form) and the current attributes of the underlying molecule. If any of these attributes are changed, the coordinates may become outdated. It is the user's responsibility to recalculate them in this case.

`dt_setbondstyles(depiction pic) => boolean`
: Sets the chiral bond styles of a depiction whose molecule contains chiral information. That is, attempts to determine a reasonable set of "in" and "out" wedges, cis and trans markings, and so forth, to accurately represent the chiral information present in the molecule. Note that this function must be applied after dt_calcxy() has been called or some other method of setting coordinates has been applied, as the bond styles depend on the particular layout of the depiction.

`dt_bondstyle(dt_Handle deph, dt_Handle ah, dt_Handle bh) => integer`
: Returns the style of bond in depiction, relative to an atom. The values that are returned are symbolic constants defined in the Depict Toolkit's header file.

`dt_ga(depiction pic, object ob) => integer`
: Returns the graphics attribute associated with object `ob` in the given depiction. If `ob` is an atom, the attribute is the one used when drawing the atomic symbol. If `ob` is a bond, the attribute is the one used when drawing the bond. If `ob` is an aromatic cycle, the attribute is the one used when drawing the circle indicating its aromaticity.

`dt_setga(depiction pic, object ob, integer ga) => boolean`
: Set the graphical attribute associated with the given object in the given depiction.

`dt_label1(depiction pic, object ob) => string`
`dt_label2(depiction pic, object ob) => string`
: Return the first and second string labels associated with the given object in the given depiction. If an error is detected returns the invalid string. String labels may be associated with any constituent of the underlying molecule, including the entire molecule itself. Depending on the type of the constituent, the label will be used in different ways. The first atom label is drawn to the lower left of the atomic symbol. The two bond labels are drawn above and below the bond. Cycle labels are unused. Labels for the whole molecule are placed above and below the depiction as a whole.

`dt_setlabel1(depiction pic, object ob, string label) => boolean`
`dt_setlabel2(depiction pic, object ob, string label) => boolean`
: Set the first and second string labels associated with the given object in the given depiction. Returns TRUE if no errors are detected.

`dt_label1ga(depiction pic, object ob) => integer`
`dt_label2ga(depiction pic, object ob) => integer`
: Returns the graphic attribute of the first and second string labels of the given object in the given depiction.

`dt_setlabel1ga(depiction pic, object ob, integer ga) => boolean`
`dt_setlabel2ga(depiction pic, object ob, integer ga) => boolean`
: Sets the graphical attributes of the first and second string labels of the given object in the given depiction. Returns TRUE if no errors are detected.

`dt_isomeric(depiction pic) => boolean`
: Returns TRUE if the depiction includes isomeric information.

`dt_setisomeric(depiction pic, boolean iso) => boolean`

Sets the isomeric depiction status of the given depiction to the given value. (A value of TRUE means that isomeric information will be drawn.) Returns TRUE if no errors are detected.

`dt_compressed(depiction pic) => boolean`
: Returns TRUE if chains of like atoms are compressed in the depiction.

`dt_setcompressed(depiction pic, boolean comp) => boolean`
: Sets the compressed depiction status of the given depiction to the given value. (A value of TRUE means that chains of like atoms are compressed.)

`dt_schematic(depiction pic) => boolean`
: Returns TRUE if the depiction is simplified to the point of a schema. (All normal carbons are suppressed, no circles in aromatic rings, etc.)

`dt_setschematic(depiction pic, boolean schema) => boolean`
: Sets the schematic depiction status of the given depiction to the given value. (A value of TRUE means that the depiction is simplified to a schema.) Return TRUE if no errors are detected.

`dt_orient(depiction pic) => integer`
: Returns the orient property of the depiction.

`dt_setorient(depiction pic, integer value) => boolean`
: Sets the orient property of the depiction. The value controls the allowed orientations of the depiction.

11.3 Conformations

A conformation is an association of 3D Cartesian coordinates with the atoms of a molecule; there is an (x, y, z) coordinate for each atom. It is a "lower level" object than a 2D depiction: A 2D depiction is assumed to be useful only in that it can be shown to the user, whereas a 3D conformation might be the basis for chemical modeling, computing charge density, energy calculations, and so forth.

Unlike a depiction, there are no mechanisms for drawing conformations (e.g. no labels, graphics attributes, etc.), but a 3D conformation can be projected on to a 2D depiction.

`dt_alloc_conformation(molecule molec) => conformation`
: Returns a conformation for the given molecule. All (x,y,z) coordinates in the conformation are set to 0. Note that there is no theoretical limit to the number of conformations that may be created for the same molecule, though there may be practical limits.

: Each conformation object has a base object: the molecule it represents (see dt_base()). If the base molecule is deallocated the conformation will automatically be deallocated and its handle revoked. Note that, unlike depiction objects, conformation objects are not deallocated when the molecule is modified.

`dt_project(depiction pic, conformation conf) => boolean`
: Sets the coordinates for the atoms of the given depiction by projecting the given conformation's atom coordinates to the x/y plane (i.e. ignore the z axis).

11.4 Modifying Depictions and Conformations

The following functions apply to both depictions and conformations.

`dt_setcoord(object ob, atom at, real x, real y, real z) => boolean`
: Sets the (x,y) or (x,y,z) coordinates of the atom in the depiction or conformation, respectively. If ob is a depiction, z is ignored but must be present.

`dt_zerocoord(object ob) => boolean`

Zeros the (x,y) or (x,y,z) cartesian coordinates of the atom in the depiction or conformation, respectively.

`dt_getcoord(object ob, atom at, RETURN real x, RETURN real y, RETURN real z) => boolean`

Gets the (x,y) or (x,y,z) cartesian coordinates of the atom in the depiction or conformation, respectively. If ob is a depiction, z is not altered but must be present. Note that x, y, and z are return values (e.g. are modified by the function).

`dt_rotate(object ob, real x, real y, real z, real theta) => boolean`

If ob is a conformation, modifies the 3D coordinates associated with the atoms in the conformation ob, by rotating them by angle theta around an axis defined by x, y, and z.

The axis of rotation is the line from the origin to the point (x,y,z). Note that its length is irrelevant, only its direction is of interest; you can think of (x,y,z) as a vector. For this reason, at least one of x, y, and z must be nonzero. The angle of rotation is given by theta and is expressed in radians. The sense of the angle is given by the "right hand rule": if you curl the fingers of your right hand and extend your thumb outward as though hitchhiking, your fingers curl in the direction of positive theta when your thumb is pointing in the vector's direction.

If ob is a depiction this function modifies the 2D coordinates associated with the atoms in the depiction by rotating them through angle theta about the origin. Since points are rotated about the origin, the values x, y, and z are ignored. The angle of rotation given by theta, expressed in radians, is positive in the counterclockwise direction.

Returns TRUE if no errors are detected.

`dt_translate(object ob, real dx, real dy, real dz) => boolean`

Modifies the (x,y) or (x,y,z) coordinates associated with the atoms in the depiction or conformation, respectively, by adding dx, dy, and dz to the x, y, and z coordinates of each atom. If ob is a depiction, the parameter dz must be present but is ignored.

`dt_scale(object ob, real cx, real cy, real cz) => boolean`

Modify the (x,y) or (x,y,z) coordinates of a depiction or conformation, respectively, associated with the atoms in ob by multiplying their values by the given factors. If ob is a depiction, cz is ignored but must be present.

11.5 The Drawing Library

The Drawing Library provides all I/O for Daylight Toolkit depictions. All actual drawing (e.g. the results of calling the function dt_depict()) is done through calls to the Drawing Library. The Drawing Library is not an intrinsic part of the Daylight Toolkit, but rather is a replaceable module: Different versions can be attached to allow the Daylight Toolkit to work in a variety of environments, such as the X Window System, serial terminals, PostScript devices, and so forth.

Several example drawing libraries are supplied with the Daylight Depict Toolkit; they can be found in `$DY_ROOT/contrib/src/depict`. If you are writing a new Drawing Library, we suggest you begin with these functioning examples.

12. Reaction Toolkit

12.1 Introduction:

The reaction toolkit provides a set of tools which support both specific and generic single-step reactions. These tools add the capability to address numerous reaction-oriented chemical information problems. These tools are integrated into the Daylight system and are used extensively within Thor and Merlin to add support for reactions to these systems.

The reaction toolkit adds support for two additional object types:

Reaction Toolkit Object Classes	
Reaction	a single-step reaction
Transform	a generic reaction

The reaction object is actually implemented within the Smiles toolkit library. The transform object is implemented within the Smarts toolkit library. Note that the reaction toolkit is licensed separately, even though the toolkits are contained within the Smiles and Smarts libraries.

12.2 Polymorphism and the Reaction Toolkit:

The extensive use of polymorphism for both reaction and transform objects is one of the key principals which makes the reaction toolkit convenient to use. A design criteria for a reaction object is that it behave as much like a molecule object as possible. Similarly, a design criteria for the transform object is that it behave like a pattern object.

In effect, a reaction object is a "superset" of a molecule object. A reaction can do everything a molecule can, and then some (which we'll cover in detail).

For example, a reaction contains one or more molecule objects. These are the components of the reaction (reactant, agent and product molecule). Each of these molecule objects in turn contains atoms, bonds, and cycles. Now one can certainly take a stream of molecules over a reaction. This works as one would expect, returning a stream which contains every component molecule in the reaction.

```
dt_stream(reaction, TYP_MOLECULE) => all molecules in the reaction
```

One can also take streams of atoms, bonds, or cycles over a reaction, effectively ignoring the molecule layer of the reaction. In this case, the streams work exactly the same for molecules and reactions.

```
dt_stream(reaction, TYP_ATOM) => all atoms in the reaction
dt_stream(reaction, TYP_BOND) => all bonds in the reaction
dt_stream(reaction, TYP_CYCLE) => all cycles in the reaction
```

Note that in the case of streams of atoms or bonds over a reaction, the resulting stream will contain ALL of the atoms, bonds or cycles in every molecule in the reaction.

Generally, the strategy for reaction toolkit programming is to ignore the "molecule layer" of a reaction whenever possible. This results in toolkit code which is most flexible in that the code will correctly process both molecules and reactions.

As an example, consider the following code:

```
#include "dt_smiles.h"
#include "dt_depict.h"

main() {
  dt_Handle ob, d, atoms, atom;
  char line[400], *msg;
  int len, count;

  /*** Get SMILES from user ***/

  if (!gets(line)) return (0);

  /*** Create object. dt_smilin returns a molecule or reaction, but we
       don't care which.  The rest of the toolkit calls operate equally
       well on either. ***/

  ob = dt_smilin(strlen(line), line);

  /*** We could check the type of object returned if we wanted, but it isn't
       necessary  (dt_type(ob) would return TYP_MOLECULE or TYP_REACTION) ***/

  count = 0;
  atoms = dt_stream(ob, TYP_ATOM);
  while (NULL_OB != (atom = dt_next(atoms)))
    if (dt_number(atom) == 6) count++;        /*** Count carbons ***/
  dt_dealloc(atoms);

  printf("The object contains %d carbon atoms.\n", count);

  /*** Note that dt_alloc_depiction(3) can take a reaction or molecule object
       in version 4.5 ***/

  d = dt_alloc_depiction(ob);
  dt_calcxy(d);

  /*** Call drawing library to show depiction ***/

  dl_beginscreen();
  dt_depict(d);
  dl_endscreen(d);

  /*** Destroy objects. ***/

  dt_dealloc(d);
  dt_dealloc(ob);
  return(1);
}
```

Whether the user enters a reaction or molecule SMILES is completely irrelevant to the program, the way it is coded, or its execution. This example program and many others like it (cansmi, showparts, protons, hbonds, smarts_filter, addfp, etc.) only need be recompiled under version 4.51 or later to be fully reaction-capable.

The other important factor which makes the reaction toolkit convenient is the treatment of derivative objects (paths, substruct, pathsets, depictions, conformations, fingerprints). Each of the derivative object types has been extended to handle Reaction objects directly. There is no need to use or understand the behavior of a bunch of new derivative objects specifically for reactions.

In the case of derivative objects, the molecule layer of a reaction is ignored; the derivative objects just work at the atom and bond layer. For example, the depiction object used in the example code above handles reactions just as well as molecules. One can create a depiction for either a molecule or a reaction object. The returned depiction objects behave exactly as in version 4.42 with one exception: the base object (dt_base(3)) of a depiction may now be either a reaction or molecule; in version 4.4 the base of a depiction was always a molecule. See section 12.7 for further discussion of derivative objects and reactions.

12.3 Processing reactions:

A reaction consists of a set of molecule objects, each has a specific role in the reaction: reactant, product, or agent. Agents are molecules which do not contribute atoms to the products, or accept atoms from the reactants. Note that this definition is not enforced by the toolkit. It is manifested in the definition of atom maps for reactions.

This section focuses on tookit functions which are specific to reaction objects or functions which have new, unique behaviors for reaction objects. These functions are generally useful for building reactions from scratch and for manipulating reaction objects.

`dt_alloc_reaction(void) => Handle reaction`

> Allocates a new, empty reaction object. This reaction will have no child molecule components.

`dt_addcomponent(Handle reaction, Handle mol, integer role) => Handle mol`

> Adds a molecule object to a reaction. The role (DX_ROLE_REACTANT, DX_ROLE_AGENT, DX_ROLE_PRODUCT) indicates the role which the molecule will take in the reaction. A copy of the molecule is added to the reaction. The original molecule is unchanged. The reaction must be in modify-on state. Returns the molecule object within the reaction to which the given molecule was added.
>
> Practically speaking, a reaction object will have at most one each of reactant, agent, and product molecules and these are generally processed (eg. streams of molecules over a reaction) in reactant-agent-product order. If one adds multiple molecule objects to a reaction with the same role, these are combined within the reaction object. The way to think about this is that molecules are used as the internal representation of structural data in a reaction, yet the reaction object reserves the right to change it's internal representation as necessary. Since the original molecules are unaffected, this works out well.

`dt_getrole(Handle ob, Handle reaction) => integer role`

> Returns the role which the object plays within a reaction. 'ob' can be an atom, bond, cycle, or molecule. Returns (-1) if 'ob' is not part of the given reaction. The role returned will be one of the contstants: DX_ROLE_REACTANT, DX_ROLE_AGENT, or DX_ROLE_PRODUCT. It is not possible to change the role of an object within a reaction. The role is set during creation of the reaction (via dt_smilin(3) or dt_addcomponent(3)) and is immutable.

There are quite a few functions which take on new capabilities when processing reactions:

`dt_smilin(string smiles) => Handle object`

> When given a reaction SMILES string, interprets the SMILES and returns a newly-allocated reaction object. Note that dt_smilin(3) returns the appropriate object (either molecule or reaction) for the given SMILES string. This behavior also depends on the licenses available:

Input SMILES	Toolkit licenses available	dt_smilin(3) behavior
Any SMILES	none	Program exits
Molecule SMILES	smiles	returns Molecule object
Molecule SMILES	smiles, reaction	returns Molecule object
Reaction SMILES	smiles	returns NULL_OB, warning in error queue
Reaction SMILES	smiles, reaction	returns Reaction object

`dt_cansmiles(Handle reaction, integer iso) => string smiles`

Returns the canonical SMILES for a reaction. When 'iso' is FALSE, returns the unique SMILES. The unique SMILES is the canonical SMILES where all agents, isomeric and isotopic information, and atom maps are ignored for generation of the SMILES.

When 'iso' is TRUE, returns the absolute SMILES for the reaction. This includes all agents, isotopic and isomeric information, and atom maps.

`dt_xsmiles(Handle reaction, integer iso, integer explicit) => string smiles`

Returns an exchange SMILES for a reaction. When 'iso' is FALSE, returns an exchange SMILES without map, stereo or isotopic information.

When 'iso' is TRUE, returns an absolute exchange SMILES for the reaction. This includes all agents, isotopic and isomeric information, and atom maps.

The 'explicit' parameter, when TRUE, returns an exchange SMILES with all atomic properties explicit in the string.

`dt_type(Handle reaction) => integer TYP_REACTION`

For a reaction object, returns the constant TYP_REACTION.

`dt_typename(Handle reaction) => string "reaction"`

For a reaction object, returns the string constant "reaction".

`dt_info(Handle reaction, string "smiles") => string input SMILES`

Returns the input SMILES string used to create the reaction object.

`dt_mod_is_on(Handle reaction) => boolean state`

Returns the modify-state for the given reaction.

`dt_mod_on(Handle reaction) => boolean ok`

Puts a reaction object and all of its component molecules in modify-on state. A reaction must be in modify-on state to add components, or modify any of the component molecules. Note that one can indirectly put a reaction in modify-on state by calling dt_mod_on(3) for one if its component molecules.

`dt_mod_off(Handle reaction) => boolean ok`

Puts a reaction object and all of its component molecules in modify-off state. Causes every molecule to be checked for structural validiity. This function fails if any of the component molecules is invalid. If the function fails, the entire reaction is deallocated.

`dt_dealloc(Handle reaction) => boolean ok`

Deallocates a reaction and all of its component molecules, atoms, bonds and cycles.

The following code gives a simple example of creation and manipulation of a reaction object. In this example, a reaction is built two different ways: first, a reaction is created from scratch, and molecule objects are added to build up the reaction. Second, a reaction is built from a single reaction-SMILES. The resulting reactions have the same unique SMILES.

```
void build_reaction(void)
{
  dt_Handle reaction1, reaction2;
  dt_Handle mol1, mol2, mol3;
  dt_String smi1 = "CCO";
  dt_String smi2 = "CC(=O)O";
  dt_String smi3 = "CCOC(CC)=O";
  dt_String smi4 = "CCO.CC(=O)O>OCC>CCOC(=O)CC";
  dt_String cansmi1, cansmi2;
  dt_Integer slen1, slen2;

  /*** Make molecule objects.  We'll build the reaction from its pieces ***/

  mol1 = dt_smilin(strlen(smi1), smi1);
  mol2 = dt_smilin(strlen(smi2), smi2);
  mol3 = dt_smilin(strlen(smi3), smi3);

  /*** Make an empty reaction.  Set it to mod on.  Add the pieces. ***/

  reaction1 = dt_alloc_reaction();
  dt_mod_on(reaction1);

  /*** Note: ethanol added twice, as reactant and agent.  This is legal. ***/

  dt_addcomponent(reaction1, mol1, DX_ROLE_REACTANT);
  dt_addcomponent(reaction1, mol1, DX_ROLE_AGENT);

  dt_addcomponent(reaction1, mol2, DX_ROLE_REACTANT);
  dt_addcomponent(reaction1, mol3, DX_ROLE_PRODUCT);
  dt_mod_off(reaction1);

  /*** The molecules are no longer needed (copies are kept by the reaction).
       We can deallocate them. ***/

  dt_dealloc(mol1);
  dt_dealloc(mol2);
  dt_dealloc(mol3);

  /*** Get the unique SMILES for the reaction. ***/

  cansmi1 = dt_cansmiles(&slen1, reaction1, FALSE);
  if (cansmi1 == NULL) return;

  /***  Make a second reaction from a SMILES.  ***/

  reaction2 = dt_smilin(strlen(smi4), smi4);
  cansmi2 = dt_cansmiles(&slen2, reaction2, FALSE);
  if (cansmi2 == NULL) return;

  /*** The two unique SMILES shold be the same.  ***/

  if ((slen1 == slen2) && (0 == strncmp(cansmi1, cansmi2, slen1)))
    fprintf(stderr, "The two SMILES are the same.  Life is good.\n");
  else
    fprintf(stderr, "The two SMILES are different.  Life is bad.\n");

  dt_dealloc(reaction1);
  dt_dealloc(reaction2);
  return;
}
```

12.4 Reaction Molecules:

Reactions are made up of molecule objects. These are *normal* molecules, with a new property, **role**, which is used to distinguish the reactant, product and agent in a reaction. Molecules within reactions have the reaction as a parent, and have a value defined for their role property, but are otherwise indistinguishable from any other molecules in the toolkit.

`dt_parent(Handle molecule) => Handle parent`
: Prior to version 4.5, a molecule never had a parent object. In version 4.5 and later, if a molecule is part of a reaction object, it's parent will be that reaction, otherwise this function will return (NULL_OB).

`dt_dealloc(Handle molecule) => boolean ok`
: Removes the molecule from its parent reaction, and deallocates it.

`dt_mod_on(Handle reaction) => boolean ok`
: For a molecule which is part of a reaction, puts both the molecule itself and its parent reaction in modify-on state.

`dt_mod_off(Handle reaction) => boolean ok`
: For a molecule which is part of a reaction, puts both the molecule itself and its parent reaction in modify-on state.

  This is identical to callind dt_mod_off(3) for the parent reaction. In effect, the toolkit treats a reaction and its component molecules as a single unit for structural modification; setting the state for either the reaction or one of its child molecules sets the state for all of them.

In general, if one is modifying molecules which are part of a reaction, it is best to perform dt_mod_on() and dt_mod_off() on the reaction object itself, rather than the component molecule(s). One can easily get confused if one attempts to set mod-on and mod-off for the component molecules in a reaction.

12.5 Atom Maps:

Within the SMILES language for reactions, atom maps are numeric atom labels. All atoms within a SMILES string with the same atom map label are associated in an atom map set.

Within the toolkit, atom maps are manipulable only as atom map sets. The toolkit takes care of interpreting the labels on input SMILES and labeling the output SMILES in a systematic way.

Agent atoms and atoms which are not part of a reaction may never be put in an atom map class. Only reactant and product atoms from the same reaction may appear in a given atom map class.

There are no requirements for completeness or uniqueness of the atom mappings over a reaction. Atom mappings are independent of the connectivity and properties of the underlying molecules. The rules for an atom maps are as follows:

- ◊ Only reactant and product atoms may belong to atom map classes. Atoms which are not part of a reaction cannot belong in atom map classes.
- ◊ An atom may be unmapped or may only belong to one atom map class at a time.
- ◊ Atom map classes must contain at least least one reactant and one product atom from the reaction.
- ◊ If either the last reactant or last product atom is removed from an atom map class, the atom map class is removed.

`dt_setmap(Handle atom1, Handle atom2) => boolean ok`

Sets the two atoms to be in the same atom map class. 'atom1' and 'atom2' must be atoms from the reactant and product of the same reaction, in either order.

If either 'atom1' or 'atom2' already belongs to a map class, the result of this operation is to merge the sets of atoms into a single map class which contains 'atom1', 'atom2', and any atoms which were previously mapped to 'atom1' or 'atom2'. For example, the following four functions, applied in any order, result in a single map class which contains atoms: r1, r2, r3, p1, p2.

```
dt_setmap(r1, p1);
dt_setmap(r2, p1);
dt_setmap(r3, p1);
dt_setmap(r1, p2);
```

If 'atom2' is NULL_OB, 'atom1' is unmapped from its current map set. That is, 'atom1' will no longer be mapped to any other atoms in the reaction. The atom map set from which 'atom1' is removed remains intact unless the atom map set becomes invalid. A map class becomes invalid if it no longer contains at least one reactant and one product atom. If the atom map set becomes invalid, all of the remaining atoms are unmapped from one-another.

`dt_getmap(Handle atom) => Handle substruct`

Returns a substruct based on the reaction containing all of the atoms in the atom map set to which 'atom' belongs or NULL_OB if atom is unmapped or is not an atom in a reaction.

`dt_mapped(Handle atom1, Handle atom2) => boolean mapped`

Tests the two atoms. If the two atoms are in the same map class, returns TRUE. Otherwise, returns FALSE. This is a convenience function. It is somewhat more efficient than performing the same operation by getting the substruct for one atom and testing the other against the substruct.

12.6 Hydrogens in Reactions:

Hydrogens in reactions are handled as with molecules (suppressed unless the hydrogen is special). With reactions, there is an additional case which will make a hydrogen special. It is often desireable (eg. 1,5-hydride shift) to store information about the location of hydrogens as part of the atom map of a reaction. Hydrogens with a supplied atom map are considered "special" and these hydrogens are not suppressed in the toolkit. These mapped hydrogens appear explicitly in Isomeric SMILES for reactions. Otherwise, atom-mapped hydrogens do not appear in canonical SMILES.

Note that the special hydrogen dt_isohydro(3) can not be part of any atom map class. Hence, this special hydrogen can never be used in place of an atom-mapped hydrogen in a reaction. Any atom-mapped hydrogens must be stored as explicit hydrogens.

12.7 Reaction Queries:

A reaction query is expressed with the SMARTS language. SMARTS has been extended with reaction and atom map query syntax. There is no separate pattern object for a reaction query. When a SMARTS is interpreted, a pattern object is returned. In effect, the pattern object takes on the additional expressive capabilities for reactions.

`dt_smartin(string SMARTS) => Handle pattern`

Evaluates the given SMARTS string and creates a pattern object from it. The SMARTS may be any valid molecule- or reaction-SMARTS.

`dt_smarts_opt(string SMARTS, integer vmatch) => string SMARTS`

Returns an optimized SMARTS string. Works correctly for both molecule- or reaction-SMARTS. If "vmatch" is TRUE and the given SMARTS string is for a reaction query, dt_smarts_opt fails. Vector matching on reaction queries is not allowed.

12.8 Reactions and other objects:

The flexibility and utility of the Daylight toolkit arises partly because of the ability to create derivative objects based on Molecules. These objects include paths, substructs, pathsets, depictions, conformations and fingerprints. Each of these objects has a specific unique purpose within the toolkit, however they all share some common features which are important for reaction processing:

◊ They all have a molecule as their base object,
◊ they all store data about the atoms and bonds in a molecule,
◊ and they all ignore other attributes of the molecule not directly related to the atoms and bonds in the molecule.

These features allowed us to directly extend these objects to handle reactions. As discussed in Section 12.3, the "molecule layer" of a reaction is ignored; only the atoms and bonds of a reaction are considered.

Hence, each of these objects is now defined as having either a molecule or a reaction as its "base" object. Otherwise, their behaviors are essentially unchanged. They still store data about the atoms and bonds in their base object, and they still ignore other non-relevant attributes of their base object (like the molecules).

Briefly, we address each of the main derivative types in the next sections and highlight their behaviors with regard to reactions.

12.8.1 Paths and Substructs:

Paths and substructs are collections of atoms and bonds, which all come from the same base object. With reactions, this behavior remains unchanged. The atoms and bonds within a path or substructure must come from the same reaction but they may be from different molecules within a reaction. For example, the following code creates a path from a reaction object, adds all of the double-bonds from the reaction to the path, and returns the path.

```
dt_Handle get_db(dt_Handle ob)
{
  dt_Handle bonds, bond, path;

  /*** Inappropriate type ***/

  if ((dt_type(ob) != TYP_MOLECULE) &&
      (dt_type(ob) != TYP_REACTION)) return (NULL_OB);

  /*** Make a path.  The base of the path will be "ob" ***/

  path = dt_alloc_path(ob);

  /*** If a reaction, ignore the molecule layer.  Only deal with the
       bonds.  If a molecule, this happens by default. ***/
  bonds = dt_stream(ob, TYP_BOND);
  while (NULL_OB != (bond = dt_next(bonds)))
    if (dt_bondorder(bond) == DX_BTY_DOUBLE)
      dt_add(path, bond);

  /*** Clean up and return ***/

  dt_dealloc(bonds);
  return (path);
}
```

Note that absolutely no consideration is given to the fact that the bonds may be in different molecules within the reaction. As long as the atoms and bonds added to a path or substruct are all part of the correct base object (the object given in dt_alloc_path(3)) this succeeds.

12.8.2 Pathsets:

A pathset is a collection of paths over the same base object. The base object may be a reaction. A pathset is returned from the SMARTS matching functions.

In this case, the pathset returned depends on the type of target used for the match function:

`dt_match(Handle pattern, Handle target, integer limit) => Handle pathset`
`dt_umatch(Handle pattern, Handle target, integer limit) => Handle pathset`

This returns a pathset with "target" as its base object. "Target" may be either a reaction or molecule. The pathset will contain one or more paths. The base object (dt_base(3)) of the pathset and all paths withing the patheset will be the target object. target object. Note that this behavior holds regardless of the type of pattern used in the query (reaction or molecule query).

The semantics for pattern matching are as follows:

Pattern	**Target**	**Result**
Molecule query	Molecule object	Molecule substructure matches
Molecule query	Reaction object	All substructure matches over entire reaction
Reaction query	Molecule object	No hits
Reaction query	Reaction object	Reaction substructure matches

`dt_vmatch(Handle pattern, Handle target, integer limit) => Handle pathset`

This returns a pathset with "target" as its base object. "Target" may be either a reaction or molecule. The pathset will contain one or more paths whose base object will be the same target object.

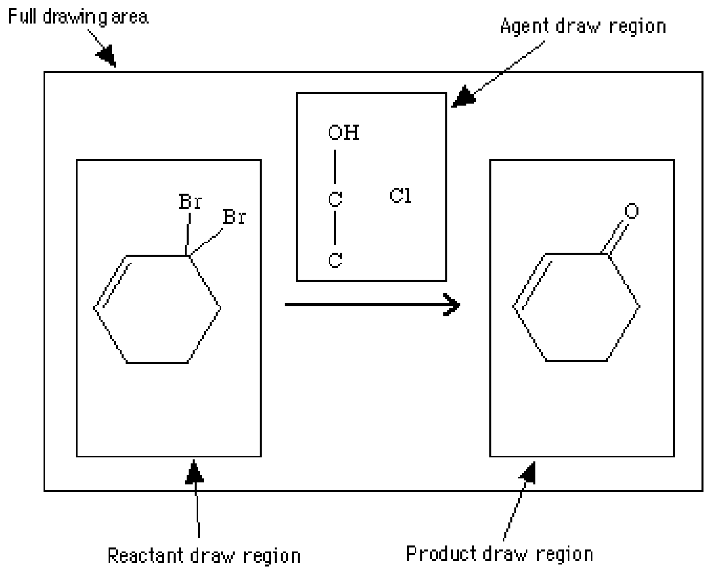
There is one important exception for vector-matching: It is only legal to use a molecule pattern for dt_vmatch(3). One may match the molecule pattern against either a reaction or molecule target, but it is not possible to use a reaction pattern for vector matching on any target (reaction or molecule).

12.8.3 Depictions:

The main distinction between a reaction depiction and a molecule depiction is the presence of a reaction arrow, and the potential desire to lay out the various reaction parts (reactant, agent, product) in different regions. These two functions are handled with dt_depict(3), and dt_calcxy(3); all other depiction-related functions remain unchanged.

`dt_calcxy(Handle depiction) => boolean ok`

Sets the coordinates for the atoms of the given depiction. In the case of a reaction depiction, it lays out the reactants, agents and products in a left-to-right orientation, with the reactants and products centered vertically and the agents shifted above the center.



If atom map classes are available for the atoms in the depiction, the toolkit will attempt to orient the reactant and product sides of the depictions the same way. The toolkit attempts to minimize the RMS distance between mapped atom pairs by reorienting the product part of the reaction depiction before laying out the parts of the reaction. This orientation first applies to ring atoms within the depiction. If no mapped ring atoms are found, non-ring atoms are used.

`dt_depict(Handle depiction) => boolean ok`

Generates the depiction, using the Daylight drawing library. For a reaction object, automatically includes a scaled arrow in the drawing. The toolkit provides no access to the arrow itself, it is drawn by the toolkit using the framega set for the depiction object.

The arrow is positioned as follows: a horizontal vector is laid out between the midpoints of the reactant and product parts of the depiction. The vector is clipped so that it doesn't overlap any parts of the reaction. Finally, the clipped vector with an arrowhead is drawn. If it is not possible to clip the vector so it doesn't overlay any part of the reaction, the toolkit will then draw a short arrow between the midpoints of the reactants and products, ignoring any overlap.

12.8.4 Conformations:

The conformation object allows the storage of (x, y, z) coordinate data for the atoms in a molecule and reaction. A conformation object makes no distinction between the roles of atoms in the reaction object. With the exception of allowing a conformation to be created from a reaction, all conformation-oriented functions remain unchanged.

12.8.5 Fingerprints:

The fingerprint object does behave differently for a reaction object versus a molecule object. The differences are seen when creating a fingerprint object, all other fingerprint toolkit functions remain unchanged. In addition, there is a new fingerprint-creation function, dt_fp_differencefp(3), which is designed primarily for reaction processing.

`dt_fp_generatefp(Handle object, integer minstep, integer maxstep, integer size) => Handle fingerprint`

Generates a fingerprint object from the given molecule, reaction, substruct, or path. For reaction objects or reaction-derived paths and substructs, the resulting fingerprint object is equivalent to the bitwise-OR of the following fingerprints:

- the fingerprint of the reactant part,
- the fingerprint of the product part,
- a bit-shifted fingerprint of the product part.

This behavior allows the fingerprint to serve as a structural screen for all superstructure-matching and allows the fingerprint to provide some discrimination power between reactant and product parts.

For reactions, the fingerprints tend to be quite dense, and are somewhat less efficient a structural screens that for molecules. The main advantage of this scheme is the full compatability of these reaction fingerprints with molecule fingerprints in the Daylight system. Note also that this fingerprint scheme doesn't provide the most appropriate measure of similarity for reactions.

`dt_fp_differencefp (Handle object, integer minstep, integer maxstep, integer size) => Handle fingerprint`

Generates a difference fingerprint object from the given molecule, reaction, path, or substruct object. This function is oriented towards reaction processing, so isn't very useful for molecules and molecule-derived paths or substructs.

For a molecule or molecule-derived object, returns the normal fingerprint, (identical to dt_fp_generatefp(3)).

For a reaction or reaction-derived object, returns the difference in fingerprint between the reactant and product parts of the object as follows:

1. Generates the count of each path in the reactant part.
2. Generates the count of each path in the product part.
3. For any paths whose count changes from reactant to product part, sets a bit in the final fingerprint.

The net result of these operations is a fingerprint of the connectivity change for a reaction. This is an extremely useful way to analyze and cluster reactions.

There is one important caveat for difference fingerprints: to work optimally, the reaction must have unit stoichiometry. If not, missing atoms on either side of the reaction will result in extraneous bits being set in the difference fingerprint.

12.9 Transforms

Transforms are very similar in behavior to patterns. Essentially the transform language is a subset of SMARTS, with some additional specific requirements. These requirements are validated on input of the transform. This also means that any valid SMIRKS is also a valid SMARTS. This also means that a SMIRKS can be optimized by dt_smarts_opt(3). A more extensive discussion of the relationship of SMILES, SMARTS, and SMIRKS can be found in the Daylight Theory Manual.

`dt_smirkin(string SMIRKS) => Handle transform`
: Interprets the given input string as a SMIRKS and creates a transform object from the SMIRKS.

`dt_smarts_opt(string SMIRKS, integer vmatch) => string SMIRKS`
: Returns an optimized SMIRKS string. Remember, SMIRKS are a subset of SMARTS. "vmatch" must be FALSE for transform SMIRKS. Optimizing a SMIRKS is useful because the first step in application of a transform object is a SMARTS-match on either the reactant or product side of the transform. Hence, the optimizations performed by dt_smarts_opt(3) are also relevant to transforms.

`dt_type(Handle transform) => integer TYP_TRANSFORM`
: For a transform object, returns the constant TYP_TRANSFORM.

`dt_typename(Handle transform) => string "transform"`
: For a transform object, returns the string constant "transform".

`dt_info(Handle transform, string "smirks") => string input SMIRKS`
: Returns the input SMIRKS string used to create the transform object.

`dt_match(Handle transform, Handle target, integer limit) => Handle pathset`
: Performs a SMARTS match, using the transform object as a pattern, and returning the pathset over the target reaction. Note that any valid SMIRKS is also a valid SMARTS.

`dt_pattern(Handle transform, integer role) => Handle pattern`
: Returns a molecule pattern object from the "role" part of the transform.

Transforms can be *applied* to molecule objects. The result of these operations is the creation of new reaction objects which contain both the starting molecules and a set of newly-created molecules. Transforms are bidirectional, they can be applied in either the forward or reverse directions. In effect, transforms represent generic reactions. Specific instances of these generic reactions can be created from the combination of a transform and a set of molecules, which act as reactants or products in the specific reaction.

`dt_transform(Handle transform, Handle som, integer direction, integer limit) => Handle sequence of reactions`
`dt_utransform(Handle transform, Handle som, integer direction, integer limit) => Handle sequence of reactions`
`dt_xtransform(Handle transform, Handle som, integer direction, integer limit) => Handle sequence of reactions`
: Applies the given transform to the molecule or sequence of molecules "som". Note that the molecule or sequence are not altered by the function. The result is a sequence of newly-allocated reaction objects, which represent specific instances of the reaction. The parameter "limit" controls whether only the first reaction found is returned or all of the possible answers are returned. The "limit" parameter has the same semantics as in dt_match(3).

  The "direction" may be one of DX_FORWARD or DX_REVERSE. When direction is DX_FORWARD, the given molecules are treated as reactants and the transform is applied in the forward direction to the molecules. When "direction" is DX_REVERSE, the given molecules are treated as products and the transform is applied in the reverse direction.

The application of a transform logically occurs in two steps. In the forward direction, the reactant side of the transform is matched, as SMARTS, against the set of molecules given. Each place where the SMARTS matches is marked. In the second step, the atom and bond changes in the transform are applied to the matched molecules.

The only difference between dt_transform(3) and dt_utransform(3) is the function which is used to match the SMARTS expression (dt_match(3) and dt_umatch(3) respectively). The net result is that with dt_utransform(3), the resulting answers are generated from the unique set of matches, while with dt_transform(3), the complete set of answers results.

Similarly, dt_xtransform(3) uses dt_xmatch(3) for the initial SMARTS match. The net result is that dt_xtransform(3) always returns exactly one new reaction. This new reaction may have more than one application of the transform within it.

A transform (at least in one direction) can be thought of as a SMARTS expression plus a set of atom and bond changes.

The resulting sequence of reaction objects are owned by the user. Both the sequence and the reactions must be deallocated by the calling program when done with them. The given molecules or sequence of molecules are not modified by the function.

The transform processing functions set atomic properties for the newly-created reaction atoms. These properties are set in order to allow the user to correlate the SMIRKS with the resulting reaction. For example, given the amide formation SMIRKS:

[C:1](=[O:2])Cl.[H][N:4][C:5]>>[C:1](=[O:2])[N:4][C:5]

and the reacting molecules:

CC(=O)Cl.NCCC

The result of this transformation will be a reaction, with the following atomic properties set:

```
USMILES:  CCCN.CC(=O)Cl>>CCCNC(=O)C
tmap:        54  1  2       541  2
torder:      65  1  2 3     *97  8   (* has a value of 10)
```

The "tmap" property is the map class for the transform atom which matched this node in the reaction. For example, the amine Nitrogens are map class "4" in the transform, hence the tmap property for the Nitrogens in the resulting reaction are set to "4".

The "torder" property is the cardinal ordering of the reaction atoms, based on the match order of the transform. Were one to reorder the reaction atoms based on this numbering, the order would correspond to the ordering of the expressions in the SMIRKS. In the example, the original SMIRKS has 10 atomic expressions total, and the "torder" properties go from 1 - 10. The value of 4 is missing because the hydrogen is suppressed in the unique SMILES.

These properties can be accessed with the following code:

```
atoms = dt_stream(result_rxn, TYP_ATOM);
while (NULL_OB != (atom = dt_next(atoms)))
  {
    tmap = dt_integer(atom, 4, "tmap");
    torder = dt_integer(atom, 6, "torder");
  }
dt_dealloc(atoms);
```

13. Program Object Toolkit

13.1 Introduction

Program objects are used to provide two-way communication with an external process, e.g., the clogp program for computing hydrophobicity for a structure represented in SMILES. Using program objects, a calling program can start an external program, send it input, receive its output, and perform other tasks while the external program remains running and ready for more input.

A number of programs supporting program objects are supplied with the release of Daylight Software. Most of these are supplied as contributed code, in the directory:

```
$DY_ROOT/contrib/src/progob
```

The programs `clogptalk` and `cmrtalk` operate as program objects (in $DY_ROOT/bin).

13.2 Using Program Objects

Program objects are normal UNIX programs, scripts, etc. which communicate through standard input and standard output using ASCII messages with a specifically defined protocol (the "PIPETALK" protocol). Any executable within the UNIX environment which adheres to this protocol can be used as a Program Object. Note that program object programs need not be Daylight Toolkit programs. There are example program objects within the "contrib/src" directory in the standard distribution.

This approach allows a program to be used like a function, but without the need to link to the object libraries underlying the program. For instance, linking a program to functions written in C (e.g. X-windows) and in FORTRAN (e.g. the MedChem library) is extremely difficult in some versions of UNIX.

This approach also avoids the high overhead associated with running external programs from files whenever their functions are needed. For instance, some users have implemented the following approach to clogp computation from a SMILES:

◊ write the SMILES to a Thor datatree file, e.g. in.tdt,
◊ exectute the clogp program via system("clogp /tmp/in.tdt /tmp/out.tdt"),
◊ open the output .tdt file and interpret the results,
◊ remove the .tdt files via system("/bin/rm -f /tmp/in.tdt /tmp/out.tdt").

Aside from all that file manipulation, this is an extremely slow method because the clogp program must initialize itself each time a computation is run (although clogp's computations are fast, its initialization is slow because it has to read in the fragment database, read in customizations, etc.) This poor perfomance is due to the one-way nature of pipe communication via the shell. Use of clogp as a program object eliminates such problems.

Program objects are created by the function dt_alloc_program() from an executable file name. Messages consist of zero or more ASCII strings and are represented in the Daylight Toolkit by a sequence of string objects. Once the calling program has created a program object, it can converse with it using messages, via dt_converse(). Program objects are deallocated with dt_dealloc().

13.2.1 Welcome and Farewell Messages

The primary type of communication with a program object is that the calling program sends a message and the program object responds with a message. There are two other situations where program objects can send messages.

A program object sends an unsolicited message when it is first invoked; this is called the "welcome message" and is obtained with dt_welcome().

All program objects must send a welcome message (although it may be empty), and all programs which allocate program objects should call dt_welcome() after a sucessful return from dt_alloc_program().

A program object also sends a message when it is terminated; this is called the "farewell message". Calling dt_converse() with a NULL_OB message terminates the program and returns the farewell message. Sending NULL_OB is like sending a program an end-of-file. Any further calls to dt_converse() will produce empty messages. The program object should still be deallocated via dt_dealloc(). It is acceptable to deallocate a program with dt_dealloc() at any time (however, the farewell message will be lost).

13.2.2 Other Special Messages

There are several properties which are useful to know for all programs. A number of special messages are defined which all program objects will respond to:

DX_PT_HELP	respond with information about how to use the program
DX_PT_PROGRAM	respond with the name of the program
DX_PT_VERSION	respond with the integral version number of the program
DX_PT_NOTICE	respond with copyright (and/or other) notices

These definitions aren't all that special, they are simply string constants which are sent to program objects, e.g. DX_PT_HELP is defined as `"Qwerty: Say HELP."` All program objects should respond to the above messages in some useful way.

You may define (and document!) such messages as needed, for instance the program clogptalk recognizes the DX_TABLE message as a request for tabluated output (DX_TABLE is defined in medchemtalk.h as "Set TABLEOUTPUT.").

It is probable that other special messages will be defined in the future. You may register messages with Daylight Support - they will be included as comments in `dt_progob.h` so we (and others) will know not to use them.

13.2.3 Program Object Toolkit Functions

`dt_alloc_program(Handle args) => Handle prog`

Allocates a program object and executes a program. The parameter 'args' is an object containing the program name and any required arguements. 'Args' must be a stream or sequence of objects which respond to dt_stringvalue(), or a single object which responds to dt_stringvalue().

`dt_welcome(Handle prog) => Handle sos`

Return prog's welcome message, i.e., its response to being executed before being sent data. All child programs which follow the pipetalk protocol write a welcome message (which may be empty), so calling programs *must* call dt_welcome() after allocating a program object.

`dt_converse(Handle prog, Handle msgob) => Handle sos`

Send strings in msgob to prog as standard input. msgob may be a string object, a sequence of string objects, or NULL_OB (means end- of-transmission). The return value is a sequence of strings containing prog's standard output response. NULL_OB is returned on error (e.g., program inaccessible).

`dt_delimiter(Handle prog) => Integer value`

Gets the delimiter property for the program object. The delimiter property will be either DX_PT_CR or DX_PT_NONE. DX_PT_CR (the default) means that returned messages from the program object are delimited by 'newline', and the message is returned as a sequence of string objects. DX_PT_NONE means that the returned messages are not delimited, and are returned as a single string object. This string object may have multiple newlines in it, and will have a trailing newline.

`dt_setdelimiter(Handle prog, Integer value) => Boolean status`

Sets the delimiter property for the program object. The delimiter property will be either DX_PT_CR or DX_PT_NONE.

13.3 PIPETALK Protocol

The "pipetalk protocol" is the communication protocol which programs must follow if they are to be successfully used as program objects. Note that the contributed examples implement this protocol, so they can be modified rather than developed from scratch.

13.3.1 Definitions

End-of-message (EOM), end-of-transmission (EOT), and send-message- list (MSGLIST) strings are defined in `dt_progob.h` as:

```
#define DX_PT_EOM       "Qwerty: Over."
#define DX_PT_EOT       "Qwerty: Over and out."
#define DX_PT_MSGLIST   "Qwerty: Say MSGLIST."
```

These definitions should not be changed. Programs should not write them on a single line for other purposes (intended to be unlikely, given the "Qwerty: " prefix).

A message is defined as zero or more strings followed by the EOM string.

13.3.2 Receiving Messages

Messages are received by reading standard input until the receipt of a line containing only the EOM string.

Note that input lines to a program object can be arbitrarily long. Programmers should be careful not to use fixed-length buffers to receive input. The Daylight contributed code directory contains examples showing the correct way for a program object to read from standard input (see $DY_ROOT/contrib/src/c/progob).

13.3.3 Sending Messages

All messages must written to standard output in this manner:

```
message contents as string followed by newline
EOT message string followed by newline
flush standard output
```

13.3.4 Initial Response to Execution

Programs must send an initial "welcome" message upon execution. The sent message may be empty (i.e., the program must send at least the EOM line).

13.3.5 Program Operation

Programs operate by receiving a message then sending a message. Each time a message is received, one message must be sent. The sent message may be empty (i.e., the program must send only EOM). (It's OK to start sending while reading.)

To prevent "deadlock", it is critical that programs never send unsolicited messages, and that they never begin their replies until the entire input message is received (i.e. the EOM message is encountered). In addition, the program must ensure that its output buffer (standard output) is "flushed" after each message, as otherwise the parent program will sit waiting forever for a message that is stuck in the child program's internal buffers.

13.3.6 Response to Special Messages

The following strings are defined in `dt_progob.h`:

```
#define DX_PT_HELP          "Qwerty: Say HELP."
#define DX_PT_PROGRAM       "Qwerty: Say PROGRAM."
#define DX_PT_VERSION       "Qwerty: Say VERSION."
#define DX_PT_NOTICE        "Qwerty: Say NOTICE."
#define DX_PT_MSGLIST       "Qwerty: Say MSGLIST."
```

On receipt of one of the first four strings, programs should respond with an appropriate message (containing help on program operation, program name, program version, and copyright notice, respectively).

On receipt of a DX_PT_MSGLIST message, programs should send a message containing all other recognized control strings. This response can be empty. Each of the supplied messages should be responded to in a sensible manner, but it is left entirely up to program to do so.

13.3.7 Program Termination

On receipt of an EOT message, programs must send their final "farewell" message (which may be empty) and go into a quiescent state awaiting EOF on standard input, at which time the program must exit. While in the quiescent state (after EOT but before EOF), the program should respond to all messages with an empty message (just EOM).

13.3.8 Naming Convention

By convention, programs which communicate via pipetalk protocol have names that end in "talk", e.g. clogptalk.

14. THOR and Merline Toolkit: Servers

14.1 Introduction: THOR and Merlin Objects

The THOR and Merlin Toolkits provide access to the THOR and Merlin "views" of databases, respectively. Although Merlin and THOR present very different views of the data (THOR's retrieval/storage versus Merlin's exploratory data analysis), the two systems share many features, and operate on the same databases. Because of this, many features of the THOR and Merlin Toolkit are presented together.

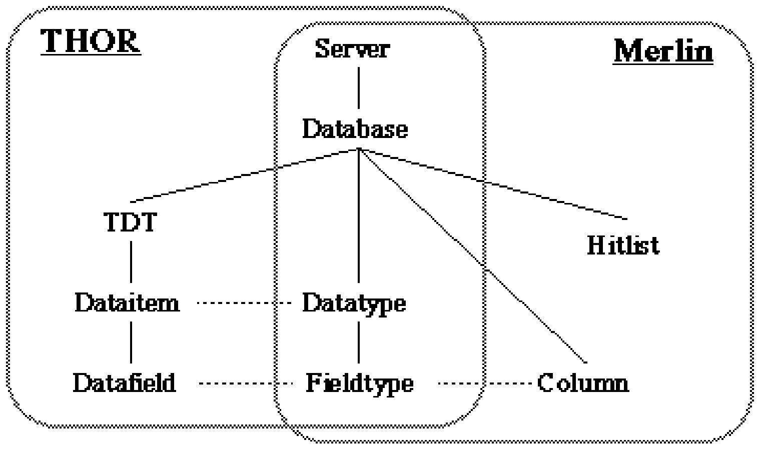
An old adage might be paraphrased here: "An example is worth a thousand words." Many tutorial examples of Thor and Merlin Toolkit usage can be found in the "contrib" directory:

```
$DY_ROOT/contrib/src/thor/
$DY_ROOT/contrib/src/merlin/
```

We **strongly encourage** you to study these examples before attempting to write Thor and/or Merlin Toolkit programs.

THOR and Merlin objects have the same basic characteristics as the objects introduced in earlier chapters. They are opaque, respond to most of the standard polymorphic functions, are subject to revocation, and so forth. However, THOR and Merlin objects have a more concrete existence since they represent real data in a database. Thus, unlike a molecule object which disappears forever when deallocated, some THOR and Merlin objects represent things that "live on" even after the Toolkit deallocates them.

The following illustrates the object hierarchies of THOR and Merlin, showing which objects are common to both:



Although this drawing shows the Server and Database objects as common to both Toolkits, there are actually different object types for each; THOR has objects of TYP_SERVER and TYP_DATABASE, while Merlin has objects of TYP_MERSERVER and TYP_POOL. Since polymorphism makes these objects behave nearly identically, we present them as a single concept. As you get into the details of their use, the differences will become apparent.

Note: By historical accident THOR server objects get the simple designation TYP_SERVER, while Merlin server objects are the more specific TYP_MERSERVER.

Below is a brief outline of each object type.

THOR and Merlin Toolkits:

- ◊ A server object represents a connection to a server. The server object "knows" about the server's network address, the interprocess- communication mechanisms, which databases the are open on the server, and so forth. Server objects are discussed in detail in this chapter.
- ◊ A database object represents an open database. In THOR this is the open database files on disk; in Merlin it is the in-memory image of the database's data.
- ◊ A datatype object represents the definition of a datatype. Datatypes define the meaning of data in a database; each dataitem in THOR is associated with its datatype (shown with a dotted line, above).
- ◊ A fieldtype object is a constituent part of a datatype object, and defines the meaning of one datafield within a dataitem; each datafield object in THOR and each column object in Merlin is associated with its fieldtype (shown with dotted lines, above).

THOR Toolkit:

- ◊ A datatree object contains a THOR Datatree (TDT) from the database.
- ◊ A dataitem object represents one dataitem from a TDT. Each dataitem object has a datatype object associated with it (shown with a dotted line, above) that defines its meaning, and has one or more child fieldtype objects for its datafields.
- ◊ A datafield object contains one datum: a string of characters. It represents the basic unit of information in the THOR system. This string of characters might be interpreted by an application as a number, text, or some other way (including binary data).

Merlin Toolkit:

- ◊ A hitlist object represents an ordered subset of the datatrees in a THOR database.
- ◊ A column object represents a "vertical slice" through the database: one datum of a particular fieldtype from each TDT in the database. For example, a column of $NAM would contain one name for each database entry.

The above brief descriptions are just a quick sketch of the object types available in the THOR and Merlin Toolkits. A detailed description of each is given in this and subsequent chapters.

14.2 Connecting to a Server

Most of the material in this chapter requires an acquaintance with the material in the Daylight Theory Manual and Daylight System Administrator's Guide. You should read these for an introduction to the concepts of server/client systems, security, search paths, file ownership, and network configuration.

THOR and Merlin are both client/server systems. The Merlin and THOR Toolkits provide a programmer's library for the client side of the client/server system -- that is, the Toolkits provide the mechanisms you need to connect to and communicate with the server side of the system. A server object represents a connection to a THOR or Merlin server. The object "knows" about the server's address, the interprocess-communication mechanisms, which databases are open on the server, and so forth.

When a client connects to a THOR or Merlin server, a new object of type TYP_SERVER2 or TYP_MERSERVER is created. The server-object is used in subsequent calls to create, open, and close databases, and other database operations, and in calls to modify security, search paths, and related administrative tasks.

A server object becomes the parent object of any databases ("pools" in Merlin) that are opened via the server (see the chapter on POLYMORPHIC FUNCTIONS for a discussion of parent objects and base objects). This implies that all such database objects can only exist while the server object exists (the actual databases, of course, exist until you erase them; here we are talking about the Toolkit objects that represent databases). If the connection to the server is broken (via dt_dealloc(server)), all databases on that server will be closed, and their child objects will be deallocated. In the case of accidental disconnection, as when a THOR or Merlin client program "crashes", the server itself closes the databases to insure no data are lost and to free allocated resources (memory, open files); if a server crashes, the client program will report "lost connection" for any subsequent database transactions.

The following functions connect to Merlin and Thor servers, respectively:

```
dt_mer_server(string host      /* server's hostname    */
              string service,   /* service name */
              string userid,    /* user's login name    */
              string passwd,    /* user's password      */
              integer isnew)    /* existing or new?     */
                  ==> Handle server

dt_thor_server(string host    /* server's hostname    */
               string service,  /* service name */
               string userid,   /* user's login name    */
               string passwd,   /* user's password      */
               integer isnew)   /* existing or new?     */
                  ==> Handle server
```

> Connects to a server on the specified host using the specified service name; returns an object of type TYP_SERVER (THOR) or TYP_MERSERVER (Merlin). If the connection already exists, returns the handle that was originally allocated for the server (see the parameter isnew, below).
>
> The parameters `host` and `service` specify the machine on which the Thor or Merlin server is running and the TCP/IP port to be used for establishing communications.
>
> The parameters `userid` and `passwd` are used by the server to verify that you have permission to connect and "log you in".

There is no built-in limit to the number of servers to which a single client can connect simultaneously. However, the operating system on which the client program is running may restrict the number of open files or otherwise limit resources in a way that limits the number of servers that can be accessed; similarly the number of clients one server can support may be restricted by the operating system of the server's machine.

Note that there is no "disconnect" function; instead, you deallocate the server object (see dt_dealloc()) to break the connection and free the resources (databases, datatrees, ...) associated with the server.

Most of the polymorphic functions behave as expected with server objects. There are several functions which are worth further discussion:

`dt_set_server_timeout(Integer value) ==> Integer old`
: Sets a connection timeout value for the toolkit. In general, it is not possible to predict whether or not a server connection will finish. It is possible for the connect functions to go away forever. In 4.61, it is possible to set a timeout, after which time the connect function will return failure. The default is not to use a timeout (timeout is zero).

`dt_getusers(Handle server) ==> Handle sos`
: Returns a sequence of strings (SOS); each string object contains the name of a currently connected user and the number of connections that user has to the server.

`dt_ping(Handle server, string text) ==> boolean ok`
: "Pings" the server: sends the text on a round-trip to the server and back to verify that the connection is working correctly.

`dt_server(Handle ob) ==> Handle server`
: This polymorphic "convenience" function works on any descendent object of a server (e.g. databases, hitlists, TDTs, etc.). It returns the server object that is the ultimate "ancestor" of the specified object. For example:

```
dt_server(database)  == dt_parent(database)
dt_server(hitlist)   == dt_parent(dt_parent(hitlist))
dt_server(fieldtype) == dt_parent(dt_parent(dt_parent(fieldtype)))
```

14.3 Security

The THOR and Merlin servers are entirely responsible for security in the Daylight system. The basic idea is that clients aren't trusted; if the servers trusted the client programs, a devious user could easily write a program to masquerade as a "proper" client, and could thereby gain access to databases. If you are concerned with security, you should never assume that client programs are legitimate; only trust the servers' security mechanisms. Client security is no security.

The THOR and Merlin servers share a single security mechanism, and usually share a single passwords file. (See the Daylight THOR-Merlin Administration Manual for how servers are started with different passwords files.) This implies that all changes that affect a THOR server (such as adding a user or changing a password) will also affect a Merlin server running on the same machine. This is also true for multiple THOR or Merlin servers running on a single machine: unless they were started with options to give each a separate passwords file, all will share changes made to any.

14.3.1 Restricted user DX_INFO_USER

A special user, DX_INFO_USER (defined in the Merlin and THOR "include" files), is restricted by the servers to a subset of the capabilities normally available. Specifically, DX_INFO_USER can't open databases or ask about security information; this user can only connect to the servers and ask what databases are available.

The purpose of DX_INFO_USER is to allow client programs to query all available servers and assemble a complete list of databases. A user can thus avoid providing the user/password login for every server, and instead only has to provide the user/password for those servers that have databases of interest.

Note that the restricted user DX_INFO_USER appears in the passwords file like any other user; if it is removed, or if a password is added for DX_INFO_USER, the "no-login" query capabilities will not be available. This allows administrators at particularly security- conscious sites to disable this feature. Thus, programmers who take advantage of the DX_INFO_USER capabilities must be prepared for it to be unavailable.

The specific Toolkit functions available to DX_INFO_USER are:

```
dt_mer_server()
dt_ping()
dt_getdatabases()
dt_exists()
dt_info()
dt_ispublic()
dt_isopen()
```

14.3.2 Adding and Changing Users and Passwords

`dt_setpassword(Handle sh, string who, string authorizing_pw, string value) ==> boolean ok`

Changes entries in the server's passwords file. This function is used to add or delete users, change their passwords, and to add "equivalent hosts" or "restricted hosts". See the Daylight THOR-Merlin Administration Manual for more information about server security.

`dt_getpasswords(Handle server) ==> Handle sos`

Returns a sequence of strings (SOS), each string-object of which contains one line from the server's passwords file. There are two types of entries in the passwords file: user/password entries, and host entries.

15. THOR and Merlin Toolkits: Databases

15.1 Introduction

Although THOR and Merlin present very different views of the data in a database, both systems present the very same data. Because of this, most operations on databases are identical in the THOR and Merlin Toolkits. This includes opening and closing databases, setting the server's "search path" security operations, and datatype-object operations. This chapter covers all of these common operations.

An old adage might be paraphrased here: "An example is worth a thousand words" Many tutorial examples of Thor and Merlin Toolkit usage can be found in the "contrib" directory:

```
$DY_ROOT/contrib/src/thor/
$DY_ROOT/contrib/src/merlin/
```

We **strongly encourage** you to study these examples before attempting to write Thor and/or Merlin Toolkit programs.

15.2 Search Path

THOR and Merlin servers maintain a "search path" -- a list of directories which are to be searched for databases (see the Daylight System Administration Manual for more details). *(Note that the search path is a property of a server, not a database. We put it in the databases chapter rather than the server chapter because it fits with other database operations.)*

Note that the directories in the search path are interpreted by the server's operating system, hence are in a format appropriate to that operating system. For example, a Macintosh client connected to a UNIX server would use UNIX syntax to specify a database path (e.g. "/thordb/mydb"). Similarly, environment variables are interpreted on the server's operating system, not the client's.

`dt_getsearchpath(Handle server) ==> Handle sos`
: Returns sequence of strings (SOS), each string-object of which contains a directory in the server's search path. The order of directories in the SOS is the order in which the directories will be searched to find a database.

`dt_setsearchpath(Handle server, string password, string path, integer replace);`
: Sets the server's search-path. You can either replace the current path, or add to it.

`dt_getdatabases(Handle server) ==> Handle sos`
: Returns a sequence of string objects (SOS) containing all databases in server's search path.

15.3 Creating and Configuring Databases

Database creation is only done by the THOR server, so the functions in this section don't apply to the Merlin Toolkit. The following functions are used to create and configure a THOR database.

15.3.1 Database Creation

`dt_thor_createdb(Handle server,int dlen, string path, int sizepri, int sizexref) => Handle database`
: Creates a new empty THOR database and opens it with "executive" permission. The parameter `path` must be a complete path, not a relative path or just a filename.

  The parameters `sizepri` and `sizexref` are the requested sizes of the primary and cross-reference hash tables, respectively. For more information about database sizes, see the reference page for this function and the Daylight THOR-Merlin Administration Manual.

15.3.2 Database Configuration

Each database can have one, two or three auxiallary databases associated with it:

◊ The datatypes database: Contains special-purpose datatype- definition TDTs (e.g. "$D_V...|". Each time a new datatype is encountered, its definition is retrieved from this database. For more information about datatypes, see the Daylight Theory Guide.
◊ The indirect-data database. Contains the expansions for indirect references. For more information about indirect data, see the Daylight Theory Guide.

Generally speaking:

◊ a datatypes database will have no associated databases
◊ an indirect-data database will have a datatypes database associated with it that defines the indirect-datatype definitions
◊ a regular chemical database will always have a datatypes database and will often have an indirect-data database.

`dt_thor_getauxillarydb(Handle database, integer type) => string path`

Returns the path (directory, filename, and suffix) of the auxillary database associated with db of type type. Type will be either DX_THOR_DATATYPESDB or DX_THOR_INDIRECTDB.

`dt_thor_setauxillarydb(Handle database, integer type, string path) => boolean ok`

Sets the database that is to be associated with db as type `type`, where `type` is either DX_THOR_DATATYPESDB or DX_THOR_INDIRECTDB.

15.3.3 Database Crunching

After a series of deletions and/or replacements, a database's data files may have "holes" in them. For example, if a TDT is enlarged (e.g. new dataitems added), it will no longer fit in its original spot; new space is allocated for it and the old space is marked "unused". THOR can sometimes re-use these available spaces (depending on the server's implementation and configuration), but generally the server is unable to make 100% use of the space in a database that has been extensively modified. This can cause a database to grow to be much larger than the amount of actual data it contains.

Crunching is the process of moving all data "forward" in the file to fill in these unused spaces, leaving all unused space at the end of the file; a pass is made through the entire database, reading and re-writing data and rebuilding the hash table. Once this is done, the file is truncated to get rid of the unused space at the end, freeing the file-system space for other uses.

The crunch operation should not be undertaken lightly, as the crunch operation is indivisible; while a crunch is under way, the server doesn't respond to other clients. Depending on database size, a crunch can take anywhere from several seconds to many minutes.

During a crunch, the database is temporarily in invalid states; for example, the hash table file is invalid until the crunch operation is complete. The database may be corrupted if some error occurs (usually an interruption such as a power failure) midway through a crunch. The actual data records may not be damaged, but hash information is usually destroyed; the data are no longer accessible. In such a case, the thordump(1) utility may be required to recover the data.

`dt_thor_crunchdata(Handle database) => boolean`

Crunches (recovers unused space from) the primary data file of a database.

`dt_thor_crunchxref(Handle database) => boolean`

Crunches (recovers unused space from) the cross-reference data file of a database.

`dt_thor_autocrunch_limit(Handle database, float limit) => float limit`

The database's "autocrunch" parameter is used to trigger an automatic database crunch whenever the fraction of free space exceeds a limit. The fraction is computed as:

```
                           bytes_free
     free space =  -----------------------
                   bytes free + bytes used
```

This function both sets *and* returns the "autocrunch" limit -- the fraction of free space which, if exceeded, will trigger an automatic crunch. The limit applies to both the primary and cross-reference data files. If `limit` is <= 0.0, the database's limit is unaffected; this serves as a way to query the current value without modifying it. Values greater than 1.0 are not permitted. A value of 1.0 will disable autocrunching.

15.4 Opening and Closing Databases

The Toolkit calls to open and close databases in THOR and Merlin are identical, but the actual operations performed by the two servers are quite different:

- ◊ THOR opens all of the database's data files (the primary and cross-reference data files, and the primary and cross-reference hash tables). These files remain open as long as the database is open. If caching is enabled (see below), data are read from the disk files into the Thor server's memory. If multiple clients open the same database, the server creates a "client context" for each, but shares the database resources (i.e. the files) among the clients.
- ◊ Merlin opens the primary data file, reads its contents into memory, and closes the file. The memory remains in use as long as the database is in use (by any user). Each client that opens the same database has its own "client context" in the server, but all clients share the database's in-memory image.

`dt_open(Handle server, string dbname, string permission, string password,RETURN integer isnew) ==> Handle database`

Opens a database on a THOR or Merlin server. The `path` is the path (directories and filename) of the database on the server machine. If it is a simple filename (no directory information), the server will search its search path for the database -- the first database found in the path that matches the name is used. If path contains any directory information, it must be a complete path - partial and relative paths are not allowed. When a complete path is specified, the server's search path is ignored.

The string `perm` is one of "r", "w", or "e", representing read, read/write, and executive permission. The `password` must be the database's password for the requested permission or higher (i.e. the executive password always works, the write password works for reading or writing, and the read password only works for reading.)

`dt_exists (Handle server, string dbname) ==> boolean isopen`

Returns TRUE if the named database exists.

`dt_isopen (Handle server, string dbname) ==> boolean isopen`

Returns TRUE if the named database is already open (either open by some other client, or marked "hold" - see dt_hold() and below).

`dt_ispublic(Handle server, string name) ==> boolean ispublic`

Returns TRUE if the named database is "public"; that is, if it has an empty read-permission password so that it can be opened without a password.

15.5 Memory Usage: Cache and Hold

15.5.1 Merlin HOLD

It can take a long time for a Thor or Merlin server to open a database: Merlin's in-memory high-speed searching requires that it scan the entire database into memory; Thor provides various levels of "caching" -- loading heavily-used parts of the database (or even all of the database) into memory to improve performance. Because of the potentially high overhead to open a database, both Thor and Merlin provide a "hold" for databases which causes the database to remain open even when no client is using it. For Merlin, "hold" means the database is retained in memory. For Thor, "hold" means the database files remain open, and cached portions of the database remain in memory.

`dt_hold(Handle database, string thorpassword) ==> boolean ok`

Marks the specified database "held", so that it will be retained in the Merlin server's memory. The password is that of the user "thor", and must be supplied even if you connected to the server as the user "thor". Returns TRUE if the operation succeeded. The operation fails if the server determines that the password is incorrect, or if database is not a Merlin database (pool) object.

`dt_isheld(Handle database) ==> boolean isheld`
: Returns TRUE if database is marked "hold". Returns FALSE if the database is not marked "hold", or if database is not a Merlin database (pool) object.

`dt_release(Handle database, string execpassword) ==> boolean ok`
: Marks the specified database "released" (not held), so that it will be removed from the Merlin server's memory when the last client closes it. The password is that of the user "thor", and must be supplied even if you connected to the server as the user "thor". Returns TRUE if the operation succeeded. The operation fails if the server determines that the password is incorrect, or if database is not a Merlin database (pool) object. Note that the database is *not* released as long as any client (including the one performing this operation) has the database open. Clients can be "evicted" to force closure; see dt_evict().

15.5.2 THOR Caching

A THOR server's performance can be improved by "caching": storing frequently-used sub-parts of the database in the server's memory. This is discussed in more detail in the Daylight Theory Manual and the Daylight System Administration Manual.

Remember that a server is free to silently ignore any and all caching requests, depending on the particular implementation and the server's configuration.

Valid caching levels are symbolic constants in the THOR Toolkit:

Thor Caching Levels	
DX_THOR_OFF	no caching
DX_THOR_RTABLE	write-through cache of hash table
DX_THOR_TABLE	complete cache of hash table
DX_THOR_RALL	write-through cache of everything
DX_THOR_ALL	complete cache of everything

The following functions control caching:

`dt_thor_cache(Handle database, int level) => boolean`
: Enable caching for the database. The parameter `level` indicates what type of caching to perform; see the table above.

`dt_thor_cachecontrol(Handle database, int when, int level) => boolean`
: Overrides cache requests from normal users; the cache-control specification becomes a property of the database, and remains in effect when the database is closed and reopened. Requires executive permission. The parameter `level` indicates how much caching to perform, as described above. The parameter `when` indicates:

DX_THOR_CACHE_NEVER
: Caching is always disabled; caching requests from other clients are prohibited and are silently ignored.

DX_THOR_CACHE_OK

Caching requests from clients are allowed; the parameter `level` is ignored. This is the default.

DX_THOR_CACHE_ALWAYS

Caching is forced whenever a database is opened, to the level specified by level; caching requests from other clients are prohibited and are silently ignored.

`dt_thor_cachesync(Handle database) => boolean`

Forces all cached data to be written to the disk immediately. This should only be done occasionally, as it is an "atomic" operation -- the entire sync is completed before any other client requests are served, which can adversely affect performance.

15.6 Database Security

There is only one function for managing the security of databases. Note that it is polymorphic; it also applies to server objects; its behavior when applied to server objects is described in the Server Security Functions chapter of this manual.

`dt_setpassword(Handle database, string what, string authorizing_pw,string newpw) => boolean`

Changes a password for the database.

Note that when a database's password is changed any existing users of that database are unaffected; a client program can keep a database open indefinitely even though the password used to open the database is no longer valid. Authorization is only checked when the database is opened.

The string what indicates which of the three passwords is to be changed; it must be one of "r", "w", or "e", for read, write, or executive passwords, respectively.

15.7 Record Locking

Thor provides a mechanism for "locking" a TDT ("record"). When a client program locks a record, the record is said to be "owned" by that client. The owner of a record has exclusive write access to that record; no other client can modify or delete that record (although they can read the record). A record can only be locked by one client at a time.

Record locking is an all-or-nothing affair: Conceptually, if record locking is enforced, then all records *must* be locked before they can be modified. In practice, if you write an unlocked record, it is automatically locked, written, then unlocked. This means if another client has that record locked, your write will fail due to a lock violation.

Once a record is locked, the client that owns the lock can do the following:

Change the record:

The client with the lock can modify the record; no other client can.

Write the record to the database:

If a modified, locked record is written to the database, the changes are "invisible" to other clients until that record is unlocked ("committed"). Other clients will "see" the original record, even though the client holding the lock sees the changes.

Delete the record:
: A deletion is essentially the same as a change: Only the owner of the lock can delete the record, and the record will appear unchanged (undeleted) to other clients until it is unlocked ("committed"). Deleting a record does **not** unlock it -- the lock remains in effect until it is explicitely removed (which causes the deletion to be "committed").

Rollback modifications:
: As long as a record remains locked, it can be "rolled back" to its original state. That is, if it has been modified or deleted, those changes are undone by the "rollback" operation. Rolling a record back does *not* unlock the record.

Commit modifications:
: When the record is unlocked, it is "committed". That is, all modifications are finalized and become visible to other clients using the database. This includes deletion -- deletions take effect when the record is unlocked.

When a record is locked by one client, all other clients that try to use the record are restricted to *read-only* operations. That is, they can only retrieve and examine the record (see dt_thor_tdtget()), and find out has it locked (see dt_thor_tdtlockedby()).

It is possible to lock a record that does not exist. This is commonly necessary when writing a new record to the database -- the record is locked, then written and finally unlocked ("committed").

The actual record locks are maintained by the Thor server. If a client disconnects from a Thor server or closes a database while it still has records locked, the locks are automatically discarded and the records are "rolled back". Any changes made but not committed are lost. Locks can only be retained while a client is connected to a Thor server and has a database open.

Record locking is not necessary in most situations. Thor's ability to merge records makes it possible for users to simultaneously modify records with little chance of conflicts. On the rare occasion when conflicts arise, Thor's timestamp facility provides adequate warning.

The following functions control locking enforcement:

`dt_thor_settdtlocking(Handle database, string password, dt_Integer enforce_locking) ==> boolean OK`
: Sets or unsets "record locking" enforcement for THOR database. If `enforce_locking` is TRUE, locking is enforced; if it is FALSE, locking is disabled.

  You can't change record locking enforcement while the database is in use (i.e. open by any other client).

  When record locking is enforced, records that are retrieved from a writeable database are automatically locked (see dt_thor_tdtget()). A writeable database is one opened with "w" or "e" permission using dt_open().

  Record locking is a permanent property of the database (i.e. it is retained when the database is closed and reopened), and it applies to all client programs using the database.

`dt_thor_tdtdttlocking(Handle database)`
: Returns TRUE or FALSE, indicating respectively that record locking is or is not enforced for the specified database.

Other functions related to or affectd by record-locking enforcement are:

dt_thor_tdtget
dt_thor_tdtget_raw
dt_thor_tdtlockedby
dt_thor_tdtput
dt_thor_tdtput_raw
dt_thor_tdtremove
dt_thor_tdtremove_raw

16. THOR and MERLIN Toolkits: Datatypes

16.1 Datatype and Fieldtype Objects

The syntax and semantics of each datum (i.e. each datafield) in a THOR database or Merlin database are defined by a datatype definition. In this chapter we examine how the THOR and Merlin Toolkits represent these datatype definitions as objects, and how to get a datatype's properties via its datatype object. Datatype definitions are discussed in detail in the Daylight Theory Manual, and the practical aspects of creating and loading datatype definitions into a database are discussed in the Daylight System Administration Manual.

A datatype object represents the definition of a datatype in object form. Datatype objects are considered a constituent part of a database or pool: They are automatically created when the database or pool is opened, and deallocated when it is closed. Datatype objects always exist for the life of the parent database or pool; they cannot be deallocated by dt_dealloc(), nor can they be copied by dt_copy().

A fieldtype object, a child of the datatype object, represents the sub-part of a datatype definition for a particular field in the datatype. For example, if a datatype defines four datafields, the datatype object will have four child fieldtype objects. Like datatype objects, fieldtype objects cannot be deallocated or copied.

If the definition of a datatype is modified while the database or pool is open (that is, the datatype-definition TDTs are re-loaded or edited), the datatype or fieldtype objects are not affected by the change; the database or pool must be closed and reopened before the change will take effect.

16.2 Getting Datatype and Fieldtype Objects

There are several methods a program can use to get datatype-object handles.

◊ A specific datatype can be retrieved by name from a database object; a stream over a database will return all datatype objects; and any object associated with a datatype (e.g. dataitems in THOR, columns in Merlin) can be asked for its datatype.
◊ Fieldtype objects can be retrieve via a stream over the datatype object, and any object associated with a fieldtype (e.g. datafields in THOR, columns in Merlin) can be asked for its fieldtype.

If you are reading through this manual front-to-back, the uses of datatype objects may not yet be apparent. Datatype objects are heavily used in the THOR and Merlin Toolkits when retrieving data from THOR and Merlin. If you are unfamiliar with how TDTs are retrieved from a THOR server, or how columns are created in a Merlin server, you should skim this material and return to it after studying the chapters on those subjects.

Functions for retrieving datatype objects and fieldtype objects are:

`dt_stream(Handle database, integer TYP_DATATYPE)`

Returns a stream of all datatypes objects in the THOR database or Merlin pool. For example:

```
dstream = dt_stream(database, TYP_DATATYPE);
while (NULL_OB != (datatype = dt_next(dstream)))
   /* do something with the datatype */
```

`dt_stream(Handle datatype, integer TYP_FIELDTYPE)`

Returns a stream of all fieldtype objects in the datatype object. For example:

```
fstream = dt_stream(datatype, TYP_FIELDTYPE);
while (NULL_OB != (fieldtype = dt_next(fstream)))
   /* do something with fieldtype */
```

`dt_getdatatype(Handle database, string tag) => datatype`

Retrieves a datatype's definition from the database db using the identifier tag. Returns a datatype object, or NULL_OB if a problem is detected. There will be a problem, for example, if there is no such datatype in db, or if the datatype's definition is badly formed.

Note that this function, called with identical parameters, will return the same handle. There is never more than one copy of a particular datatype object.

`dt_datatype(Handle obj) ==> Handle datatype`

Returns an object's datatype. Works on dataitems and datafields (THOR), or columns (Merlin).

`dt_fieldtype(Handle obj) ==> Handle fieldtype`

Returns an object's fieldtype object. Works on datafields (THOR), or columns (Merlin).

Functions for retrieving datatype properties are:

`dt_dfnorm(Handle obj, integer norm) ==> boolean isnorm`

Tests the object's normalization against "norm"; returns TRUE if "norm" is one of the object's normalizations. The object can be a datafield or fieldtype (THOR), or a column or fieldtype (Merlin). The detailed definitions of these normalizations are discussed in the Daylight Theory Manual; the following is a brief synopsis:

DX_THOR_AUTOGEN	generate second datafield from this
DX_THOR_USMILES	unique SMILES
DX_THOR_USMILESANY	unique SMILES, not TDT's root
DX_THOR_ASMILES	absolute SMILES
DX_THOR_ASMILESANY	absolute SMILES, not TDT's root
DX_THOR_GRAPH	convert SMILES to GRAPH
DX_THOR_MAKEGRAPH	produce a GRAPH subtree
DX_THOR_WHITE0	zap all spaces
DX_THOR_WHITE1	compress 2 or more spaces to one space
DX_THOR_WHITE2	compress 3 or more spaces to one space
DX_THOR_UPCASE	convert lowercase a-z to uppercase A-Z
DX_THOR_DOWNCASE	convert uppercase A-Z to lowercase a-z
DX_THOR_NOPUNCT	remove all punctuation
DX_THOR_SOMEPUNCT	remove some punctuation

DX_THOR_CASNUM	insert hyphens, verify checksum
DX_THOR_D3D	compute 3D hash
DX_THOR_REGEXP	must match regexp
DX_THOR_SMILES_NTUPLE	SMILES-ordered n-tuple data
DX_THOR_BINARY	binary data
DX_THOR_READONLY	field can't be set by user
DX_THOR_NUMERIC	field is numeric
DX_THOR_INDIRECT	indirect data field

`dt_dfnormdata(Handle obj, integer norm) ==> string normdata`

If a normalization has extra data (i.e. DX_THOR_REGEXP, DX_THOR_INDIR, DX_THOR_SMILES_NTUPLE), returns a string containing that data.

`dt_name(Handle obj) => string name`
`dt_briefname (dt_Handle obj) => string briefname`
`dt_summary(Handle obj) => string summary`
`dt_description(Handle obj) => string description`

These functions return an object's name ("verbose tag"), brief name, summary, and long description, respectively. They apply to datafield or fieldtype objects (THOR), or to column and fieldtype objects (Merlin).

`dt_tag(Handle obj) ==> string tag`

Returns the internal tag (e.g. "$SMI") of and object; works on datatypes and fieldtypes; in THOR also works on dataitems and datafields; in Merlin also works on columns.

17. THOR Toolkit: THOR Datatrees

Previous chapters discussed those aspects of the THOR Toolkit that are common with the Merlin Toolkit: servers and security, databases and datatypes. In this chapter, we will cover the THOR-specific capabilities of the THOR Toolkit.

17.1 THOR Streams

Streams are heavily used throughout THOR, in a way very analogous to their use in molecule objects. Before moving on to the details of datatree, dataitem and datafield objects, we will spend a few words discussing streams; methods for using the other THOR objects will be more apparent once the role of streams is clear.

When working with molecule objects, one usually needs to access the constituent parts (atoms, bonds, and cycles). The function dt_stream() is used for this purpose; for example, to get the atoms of a molecule, one invokes dt_stream(mol, TYP_ATOM). All of the constituent parts of a molecule are accessed this way; there is no other mechanism.

THOR objects' constituent parts are accessed the same way. For example, datafield objects are the constituent parts of a dataitem; a stream created by dt_stream(di, TYP_DATAFIELD) will contain all of the datafields in that dataitem.

Below is a description of the behavior of dt_stream() when applied to THOR objects. Although some of these object types have not yet been formally introduced, they are all presented here for completeness. If this is the first time you are reading through this material, you should skim through it just to get the general idea, then return later for a more thorough reading.

`dt_stream(Handle thor_ob, int typeval) => stream`

A stream over a THOR object contains the constituent parts that are of the specified type. The parameter `thor_ob` becomes the stream's base object (see dt_base()). If the base object is modified, the stream is deallocated and its handle is revoked.

The stream's contents will depend on the type of `thor_ob`; in the following, assume that datafield is an object of TYP_DATAFIELD, dataitem is TYP_DATAITEM, and so forth:

`dt_stream(datafield,typeval)`
: Returns NULL_OB for any typeval; there are no constituent parts to a datafield object.

`dt_stream(dataitem,typeval)`
: Returns a stream of the datafields in the dataitem when typeval is TYP_DATAFIELD; NULL_OB for any other value of typeval.

`dt_stream(datatree,typeval)`
: If typeval is TYP_DATATREE, returns a stream containing all of the subtrees in the TDT object. Each subtree object is itself a TDT to which dt_stream() can be applied.

If typeval is TYP_DATAITEM, returns a stream containing all of the dataitems attached directly to the TDT object (i.e. those data associated with the root identifier, but not dataitems that are part of subtree objects). The first dataitem object in the stream is always the root identifier itself; subsequent dataitem objects are data about the root identifier.

If typeval is TYP_ANY, returns a stream containing both the dataitem objects and the subtree objects.

Any other value of typeval will return NULL_OB.

`dt_stream(datatype,typeval)`
: Returns NULL_OB; there are no constituent parts to a datatype object.

`dt_stream(database,typeval)`
: (See the advice below regarding this use of dt_stream())

If typeval is TYP_DATATREE, returns a stream of all TDTs in the database. Such streams are unusual in that the object returned is created as it is needed. And unlike other types of streams, if a stream of TDT objects over a database is reset, it will re-create the objects if they have been deallocated. This allows you to get an object from the stream, operate on it, then discard it before moving to the next object. Without this behavior, it would be impossible to use streams over a database, as the number and sizes of the objects is often quite large.

If typeval is TYP_STRING returns a stream of string objects containing the lexical representation of all TDTs in the database (indirect data are not expanded). Like streams of datatree objects, the string objects are created "on demand"; you must take care to deallocate them as you go. Furthermore, if you reset a stream of string objects over a database, you will get different objects the second time through. Each call to dt_next() causes a new string object to be created. No other stream in the Daylight Toolkit behaves this way (i.e. streams ordinarily return the same objects the second time through).

If typeval is TYP_DATATYPE, returns a stream of all datatype objects defined for the database. This type of stream is quite ordinary compared with the previous two, and behaves like "normal" streams in the Daylight Toolkit.

`dt_stream(thorserver,typeval)`
: If typeval is TYP_DATABASE, returns a stream of all database objects that have been opened by this client on the server. Any other value of typeval will return NULL_OB.

The details and subtleties of the use of streams will become more apparent as we describe each THOR object type and the functions that operate on it. For now, simply keep in mind that streams over THOR objects work in a manner very parallel to their use in the SMILES Toolkit: They return the constituent parts of THOR objects.

You may have noticed that a stream can be formed that contains every TDT object in a database; this might tempt you to use this functionality to turn THOR into a searching system rather than its usual use as a look-up system (e.g. "Read through the database and find thus-and-such..."). This, in general, is a bad idea.

THOR's power comes from its ability to handle ambiguous identifiers, to look up TDTs very quickly, and its use of SMILES and hash tables. It is not designed to search through the entire database as one might a relational database. Although THOR streams certainly give you the power to do exactly this, it would be a poor use of THOR. Daylight's Merlin Toolkit is designed for searching; use it instead of THOR for such tasks.

The purpose of a stream that contains all the TDTs in a database is to provide a way to dump the database's entire contents; dt_stream() is ideally suited to this use.

17.2 Datatree Objects

In this section, we get to the heart of the matter: data storage and retrieval via TDT objects. The actual data stored in a database is accessible through these objects.

17.2.1 Creating Datatree Objects

The primary way to create TDT object is to retrieve it from a database using dt_thor_tdtget(). There is no way to create a TDT "from scratch", i.e. there is no such function as `dt_alloc_tdt()`. Instead, dt_thor_tdtget() has a parameter that will cause a new TDT to be created if it can't be found in the database. The idea is that you don't want to allocate an empty datatree for an identifier that already exists in the database, as you would very likely overwrite the existing data. By using dt_thor_tdtget() to create new TDTs, you are forced to examine the database first, hence are less likely to lose data.

SMILES is usually the root (main topic) of a TDT, but there is no requirement that this be the case. If you have data for a non-SMILES identifier and don't know the structure of the molecule (or the identifier is not for a molecule, i.e. "pine tar"), you can create a TDT with a non-SMILES root identifier.

THOR goes to considerable trouble to create a SMILES root for a TDT whenever possible For example, if you create a TDT with a $CAS number as its root and a $SMI subtree, THOR will recognize the $SMI and will invert the tree, making the $SMI part the root. Similarly, if you create a TDT with absolute SMILES in it but no unique SMILES, THOR will convert the absolute SMILES to a unique SMILES and put that at the tree's root. If you do store a non-SMILES-rooted TDT, then later store a SMILES-rooted TDT with the former identifier as a subtopic, THOR will automatically merge the data from the former TDT into the new SMILES-rooted TDT.

THOR does not allow non-SMILES-rooted TDT to have any subtopics (subtrees). The idea is that only structure (e.g. SMILES) is a valid main topic, as it is the only non-arbitrary identifier that works in this role. You can create such a TDT, and (as described above) THOR will try to find a SMILES somewhere in it, and rearrange the TDT to put a SMILES "on top". But if the attempt to rearrange doesn't result in a SMILES-rooted TDT, the TDT can have no subtrees, only data about the root (non-SMILES) identifier. Note that this error may not be detected until you try to write the TDT to its database (see dt_thor_tdtput(), below).

There is no way to move a TDT directly from one database to another; to do so, you must fetch a TDT from one database, create a TDT with the same identifier in the second database, then copy information from the former to the latter. There are several reasons for this restriction, the most important being that datatypes are a property of a database, so moving a TDT from one database to another could change the meaning of the data (in the case where the datatypes are defined differently).

A second function used to create TDT objects is dt_thor_str2tdt(), which will convert a string (lexical) representation of a TDT into an object. Some of the protection afforded by dt_thor_tdtget() is bypassed by dt_thor_str2tdt(): You can choose to merge the data in the string with existing data or to ignore data in the database. The latter strategy is useful when re-loading a database from a "dump" (e.g. when there is certain to be no conflict between incoming data and existing data) as it is significantly faster, but should not be used in general. Some additional protection is provided by the "timestamp" mechanism described below.

If record locking is enabled, fetching a "writeable" TDT also *locks* the TDT -- it gives the client who fetched the TDT exclusive write access until the TDT is explicitely unlocked. For a complete discussion, see the section on record locking in the Databases chapter.

17.2.2 Destroying Datatrees and Datatree Objects

There are two distinct operations that, at first glance, might seem similar. Both get rid of TDTs, but in very different ways:

> dt_dealloc(tdt) deallocates a TDT object from the client program's memory, but does not affect the database at all (in particular, the TDT object can be re-created by re-reading it from the database).
>
> `dt_thor_tdtremove(tdt)` removes a TDT from the database, but does not affect a TDT object that might represent that same TDT (in particular, if the TDT object exists, it is unaffected and can be written back to the database even though it was deleted from the database.

It is important to remember that these operations are entirely separate.

17.2.3 The Datatree Memory

Every TDT object that you create becomes a child object of its database (all functions that create TDTs have a database as their first parameter). Thus, THOR remembers each TDT object until it is explicitly deallocated or the database is closed.

Furthermore, THOR goes to some trouble to prevent the creation of two TDT objects with the same root identifier. If, for example, you try to read a TDT from a database using dt_thor_tdtget(), THOR will first check all of the child objects of the database to see if it already has that TDT; if so, you will get the existing TDT rather than a new one. The reason for this behavior is, once again, to prevent the accidental loss of data. You rarely want two TDT objects that represent the same actual TDT, as this would inevitably lead to one's modifications overwriting the other's.

If you have a TDT object that you suspect needs "refreshing" (i.e. to be re-read from the database) because it is out of date (see "Timestamps" below), it is necessary to deallocate the TDT before invoking dt_thor_tdtget(). As long as your copy of a particular TDT object exists, THOR will not re-read it from the database.

The function dt_thor_str2tdt() allows you to circumvent some of the protection afforded by dt_thor_tdtget(). Using it, you can create a TDT for an identifier that already has been fetched from the database, resulting in two object for the same main topic. Clearly this is a situation to be avoided.

17.2.4 Writing TDTs to a Database

The functions dt_thor_tdtput() and dt_thor_tdtput_raw() write a TDT or string to a database, respectively. Note that writing doesn't happen automatically when you modify a TDT object. That is, modifying a TDT *object* has no effect on the *original data in the database*: you can discard the TDT object if you like and the original data will be unaltered. Only when you invoke one of the two aforementioned functions is the database actually altered.

A write operation can be affected *record locking*. For a complete discussion, see the section on record locking in the Databases chapter.

17.2.5 Timestamps

The first time you write a TDT to a database, a timestamp is automatically added to it by the THOR server. On each subsequent write of a TDT, the server checks its timestamp to insure that it agrees with the one in the database. If the two match, then all is ok. But if they don't, the assumption is that some problem has occurred.. Possible causes for this situation include:

◊ Two clients could have retrieved the same TDT and modified it. The first client wrote it back successfully, causing the timestamp to change; the second client encountered an error as it attempted to write its version of the TDT.
◊ One client managed to create two versions of the same TDT. This is always due to the use of dt_thor_str2tdt(); the pitfalls of this function are outlined in its description below.
◊ A client used dt_thor_str2tdt() to create a TDT for an identifier that was already present in the database. TDTs created this way have a bogus timestamp of "eons ago", so they always appear to be out of data compared to existing data.

Timestamps are intended to serve as a partial protection against inadvertently overwriting data. The only way to circumvent the protection they provide is with a "forced write" of data (see dt_thor_tdtput()).

17.2.6 Merging Datatrees

As noted above, THOR detects an error when two clients attempt to modify a single TDT simultaneously. Timestamps will normally prevent one client from unknowingly erasing the changes made by the other client. But merely preventing the second client from writing its data doesn't solve the problem, since presumably the second client's data are important too.

To solve this problem, and to provide a mechanism for merging two databases into one, THOR provides a TDT merge operation. "Merging" is the process of identifying the set of unique dataitems from two TDTs and producing a single TDT from that unique set.

There are several points at which you can merge datatrees. First, when creating a TDT from its lexical (string) representation (dt_thor_str2tdt()), you can choose to merge the data from the string with any existing data as the TDT is created. Second, a TDT can be merged as it is being written to the database (dt_thor_tdtput()). And third, two TDTs can be merged into one at any time (dt_thor_tdtmerge()).

Merging is mostly useful when adding new dataitems to a TDT; it has unpredictable behavior when dataitems are deleted or modified. Consider the following examples:

- ◊ Two clients simultaneously read a TDT, and each adds one new dataitem to it. If these clients use the "merge" operation as they write the data, the resulting TDT will contain all of the original data (which was common to both client's version of the TDT) plus the two new dataitems (one from each TDT). It will be exactly as though one client had added both new dataitems, which is what we desired.
- ◊ Two clients simultaneously read a TDT, and each deletes a different dataitem. The first client writes its TDT out without trouble. But when the second client writes (with merging) its version of the TDT, the TDT will be restored to its original form, since the dataitem each client deleted still appears in the other's version of the TDT (i.e. the union of the two smaller sets is the original set). This is not what we desired.
- ◊ A single client reads a TDT, modifies one datafield, then uses the "merge" operation when writing it back to the database. Since the modified dataitem is no longer identical to the original, both versions of the data (the original and the modified) appear in the database. This is probably not what we desired (when modifying a dataitem, it is usually because it is wrong, and we want to overwrite the original data rather than merging the old with the new).

These examples should illustrate both the uses and the pitfalls of the merge operation. To briefly summarize, merging is primarily useful when you are adding data to a database, or when you are merging data from two or more databases into a single database. It is almost never useful when you are changing or deleting data.

17.2.7 Cross-Referencing

One of THOR's most important capabilities is that it can know a compound by many names (identifiers), and it can retrieve the compound using any identifier that is known. THOR achieves this by a cross-referencing mechanism: The identifier for each subtree of a SMILES-rooted TDT is stored in a secondary cross-reference database. Given any known identifier, the SMILES for all TDTs in which that identifier appears can be retrieved with a single access to the secondary database. (Recall from the THOR section of the Daylight Thoery Manual that a particular identifier may appear in several TDTs).

Cross-referencing is done automatically by the THOR server; you can not directly create cross references. Each time you write a TDT to its database, the server examines the TDT, extracts all of its identifiers, and creates a cross reference entry to the SMILES that is the root of the TDT for each one. Note that since non-SMILES- rooted TDTs aren't allowed to have subtrees, there is never a need to cross-reference an identifier to anything but a SMILES (for example, there will never be a cross-reference between a CAS number and a Wisswesser Line Notation).

The function dt_thor_xrefget() is the mechanism by which cross- reference information is retrieved. It returns a sequence of string objects, each containing a SMILES.

17.2.8 Functions on TDT Objects

`dt_thor_tdtget(Handle parent, Handle dt, string id, boolean writeable, RETURN boolean isnew) => tdt`

Gets or creates a THOR Data Tree (TDT) object from the object `parent`. The TDT's root identifier will be the identifier/datatype represented by `id` and `dt`, respectively, with `id` standardized according to the specifications in the datatype object `dt`.

The "parent" object can be either a database, or can be the root of a TDT. In the former case, the TDT is retrieved from the database and is a "root" TDT. In the latter case, a subtree is retrieved from an existing TDT object.

The parameter `writeable` indicates whether modifications to the TDT object are to be allowed. If the database is open read-only, then `writeable` must be FALSE. For a database open with "write" permission, you can choose to retrieve a TDT as "read-only". When record-locking is in effect (see Record Locking), this also controls whether a record is locked or not: Any TDT that is retrieved with `writeable` TRUE is automatically locked for exclusive access.

The parameter writeable also controls whether a new TDT can be created: If writeable is FALSE and the requested TDT is not in the parent, no TDT object is created and the function returns NULL_OB. If `writeable` is TRUE and the TDT doesn't exist, a new TDT object is created representing a TDT that is not yet in the database.

`dt_thor_xrefget(Handle database, Handle datatype, string id, RETURN int iserror) => sequence`

Gets a cross-reference sequence from a database, or NULL_OB if the identifier doesn't appear in any SMILES-rooted TDT or if an error is detected. If NULL_OB is returned, the return parameter `iserror` is TRUE if it was due to an error.

A cross-reference sequence contains a set of string objects. The first string object in the sequence contains the original identifier that you asked about. The second through last string objects each contain the SMILES of a TDT in which the specified identifier appears. For example, a request with the datatype object for "$NAM" and id "dichloroethene" might yield a sequence with three string objects, respectively containing `"dichloroethene"`, `"ClC=CCl"`, and `"ClC(Cl)=C"`.

`dt_thor_tdtput(Handle tdt, boolean merge) => integer`

Writes a TDT to its database (see the THOR-specific description of dt_parent(), below); modifies the timestamp dataitem to reflect the current time.

If the THOR server detects that the TDT is out of date (its timestamp is older than that of the same TDT in the database), then the parameter merge indicates what is to be done:

- If merge is FALSE, the write simply fails. Note that this function's return value (described below) clearly distinguishes between out-of-date timestamps and other failures, so it is possible to recover gracefully from these unlikely collisions between clients.
- If merge is TRUE, the data from the database are merged with the data in TDT, the timestamp of TDT is changed to the that of the data from the database, and the TDT

is written to the database.

The function returns the following values:

1 == Successful write: The TDT object was written to the database, and no problems were detected.

0 == Out-of-date timestamp: The timestamp of the TDT was out of date and merge was FALSE, or another client wrote the TDT as the merge was in progress. The TDT is not stored.

-1 == Error: Some problem was detected (invalid or revoked handle, database is a virtual database, database is NULL_OB, error communicating with the server, etc.). The TDT is probably not stored.

This function can also have some rather dramatic effects on the structure of the TDT itself, including changing the very datatype and identifier of TDT's root. See the manual page for details.

`dt_thor_tdtremove(Handle tdt) => integer`

Permanently removes (erases) a TDT from the database. Does not affect the TDT object (i.e. it only affects data in the database, not the TDT object itself).

`dt_thor_tdt2str(Handle tdt, boolean expand) => string`

Converts the TDT object into its lexical (string) representation. If `expand` is FALSE, indirect references in the datatree are not expanded; the string representation will contain the "raw" indirect reference identifier. If `expand` is TRUE, indirect references are expanded.

`dt_thor_str2tdt(Handle database, string tdtstr, boolean merge) => tdt`

Converts tdtstr, the lexical (string) representation of a TDT, into a TDT object associated with the database db.

If `merge` is FALSE, a TDT object representing only the data from `tdtstr` will be created. Note that without the merge operation, it is possible to create a TDT object with a root identifier and datatype that "conflicts" with one in the database or with an existing TDT object (i.e. the same root identifier/datatype but with different data).

If `merge` is TRUE, a TDT object representing data from the database, merged with data from the string, is created.

`dt_thor_tdtmerge(Handle tdt1, Handle tdt2) => tdt1`

Merge the dataitems from the TDT object `tdt2` into the TDT object `tdt1`; deallocate the object `tdt2`. When the merge is complete, `tdt1` contains the set of all unique dataitems from both `tdt1` and `tdt2`, and its timestamp is the newer of `tdt1` or `tdt2`.

`dt_thor_tdtput_raw(Handle database, string tdtstr)`

Puts a TDT string directly into the database without creating a TDT object, without any normalization, and without any of the safeguards that go with normalizations. Very fast and very dangerous.

This function is designed to be used only for re-loading data that was dumped (see dt_stream(db, TYP_STRING)) from an existing Thor database that was created using the exact same release of the software.

`dt_thor_tdtrevise(Handle tdt) => boolean ok`

Revises a TDT in preparation for storing it in a Thor database. This function is normally only used internaly by dt_thor_tdtput(), but can be called directly if needed. See the description of dt_thor_tdtput() for an explanation of the changes that this causes.

17.3 Dataitem and Datafield Objects

Previous sections described how to open a database and how to get and store a TDT; this section describes how to retrieve and modify the actual data contained in a TDT.

Most of the functions that access and modify a TDTs contents are of the "polymorphic" variety, so there are only a few new functions to be introduced here. In particular, you access the dataitems within a TDT and the datafields within a dataitem using dt_stream(), you access and change the value of a datafield using the same functions that work on string objects, and you find the "verbose tags" (the definitions for each datafield) using dt_tag(), dt_name(), dt_briefname(), dt_summary() and dt_description(). See the datatypes chapter for more information.

The following C code fragment illustrates this idea:

```
/* The outer loop is over all dataitems in the TDT.  Note that
   it skips subtrees and their dataitems; we will only see the
   root identifier (1st dataitem) and the data attached to it.
   The inner loop is over the datafields of each dataitem */

char *field, *label
dt_Handle di_stream, di, df_stream, df;

di_stream = dt_stream(tdt, TYP_DATAITEM);
while ( (di = dt_next(di_stream)) != NULL_OB) {
    df_sdtreamream(di, TYP_DATAFIELD)
    while dt(dnfext=(df_stream)) != NULL_OB) {
        dfiestdringvalue(df);
        dlabeinfo(df, strlen("name"), "name");
       /* do something, like print the datafield's label and value.*/
        }
    }
```

17.3.1 Functions on Dataitems and Datafields

dt_thor_alloc_dataitem(Handle tdt, Handle datatype) => Handle dataitem

Allocates a new dataitem object in `tdt`. The dataitem is created with the number and types of fields specified by datatype; the datafields initially contain empty strings (note that these empty strings are not the invalid string -- they are valid strings with no characters in them). Returns a handle to the dataitem, or NULL_OB if a problem is detected.

dt_datatype(Handle dataitem) => Handle datatype

Returns the datatype object associated with dataitem.

dt_dfnorm(Handle datafield, integer norm) => boolean isnorm

Tests the datafield against the given normalization; returns TRUE if that is one of the datafield's normalizations. Normalizations are defined in the Toolkit's header files; for example:

```
dt_dfnorm(df, DY_THOR_BINARY) => TRUE if df is binary data
dt_dfnorm(df, DY_THOR_READONLY) => TRUE if df is read-only
```

dt_dfnormdata(Handle datafield, integer norm) => string normdata

If a normalization has extra data (i.e. REGEXP, INDIR, SMILES_NTUPLE), returns a string describing that data. For example, if a datatype is defined as:

```
$D_V_N|
```

Then the call dt_dfnormdata(df, DY_THOR_INDIRECT) would return "$I" when df was the second datafield of a CLOGP dataitem.

`dt_thor_raw_datafield(Handle datafield) => string fieldvalue`

Returns the "raw" value of a datafield. Datafield's contents are normally accessed using dt_stringvalue(), which automatically expands indirect data. This function is like dt_stringvalue() except that it returns indirect datafields without expanding them. For datafields that are not indirect, it is identical to dt_stringvalue(datafield).

`dt_thor_moveitem(Handle moveh, Handle afterh) => boolean ok`

Moves the object `moveh` from one place to another in the TDT; in particular, to after the object `afterh`. There is no way to move an object to before the first object in a TDT, as this would replace the TDT's identifier, nor is it possible to move the first object in a TDT (its root identifier) somewhere else. The objects `moveh` and `afterh` must have the same root TDT (although one or the other may be part of a subtree; see below).

There are four possible combinations of object types for objects moveh and afterh (where "==" is shorthand for "is of type"):

1. `moveh == TYP_DATAITEM, afterh == TYP_DATAITEM`
   This move is always legal; the dataitem moveh is moved so that it follows afterh . The two dataitems may be in different subtrees or the root tree to begin with; moveh ends up in the subtree of afterh (it is re-parented).
2. `moveh == TYP_DATATREE, afterh == TYP_DATATREE`
   This is legal if both moveh and afterh are subtrees; moveh is moved so that it follows afterh.
3. `moveh == TYP_DATATREE, afterh == TYP_DATAITEM`
   If afterh is a dataitem of the root of the tree, moveh becomes the first subtree of the TDT. If afterh is a dataitem in a subtree, moveh is moved to after the subtree that afterh is part of.
4. `moveh == TYP_DATAITEM, afterh == TYP_DATATREE`
   This combination is never legal; it doesn't make sense.

Returns TRUE if the move is completed successfully; FALSE if errors are detected. If the move is illegal, the TDT is unaffected. If the move is legal but fails, the TDT will probably be corrupt (such failures are generally caused by corrupt TDTs being passed in; this almost never happens).

18. Merlin Toolkit

18.1 Introduction

Previous chapters discussed those aspects of the Merlin Toolkit that are common with the THOR Toolkit: servers and security, databases and datatypes. In this chapter, we will cover the Merlin-specific capabilities of the Merlin Toolkit.

Merlin uses a "spreadsheet" model to represent a database. This is discussed in greater detail in the Daylight THOR-Merlin Administration Guide. Merlin has two objects, the hitlist and the column that represent this view of the database. Two other concepts, the row and the cell, are also important, but there are no row or cell objects in Merlin.

- ◊ A column object represents a "column" of data from a database, i.e. one specific field from each TDT in the database. A column is the "y axis" of the "spreadsheet" view of the database.
- ◊ A "row" is the "x axis" of the spreadsheet view of the database; it is data from a single TDT. There is no row object; it is just an idea we use to convey the workings of Merlin.
- ◊ A hitlist object represents an ordered set of rows from a database. That is, the object holds a set of "hits" (rows) and a particular ordering of those rows. Various search operations affect which rows belong in the hitlist, and various sort operations affect the order of the rows in the hitlist.
- ◊ A "cell" is the data at the intersection of a row and a column. There is no cell object.

18.2 Tasks -- "Time Slicing"

Although Merlin is quite fast at searching and sorting, certain tasks can take a significant amount of time. Since the server has to serve many clients, tasks that take a long time have to be "sliced" into smaller time segments so that requests from various clients can be interleaved. This prevents any one client from "hogging" the server for a long time. In addition, a client can abort a time-sliced task part way through.

All sorting and searching Toolkit functions are time-sliced. These functions, and the function dt_continue() (described below), have a "status" parameter indicating how the task is progressing:

`DX_STATUS_IN_PROGRESS`	not finished, task in progress
`DX_STATUS_DONE`	finished search, target found
`DX_STATUS_NOT_FOUND`	finished search, target not found
`DX_STATUS_ERROR`	error, operation not completed

The following three functions are used in conjunction with the searching and sorting functions (which are described in detail below) to carry out time-sliced functions:

`dt_continue(Handle server, RETURN integer status) ==> integer` progress
: Continues the current task in progress. Returns the progress on the task; dividing this value by the value returned dt_done_when() will yield the fraction of the task that is completed. You can only call this function when a task is in progress, e.g after a search or sort function has returned a status of `DX_STATUS_IN_PROGRESS`.

`dt_abort(Handle server) ==> integer ok`
: Aborts the current task. A server can only have one task in progress for any particular client, so starting a second task (another search or sort) also has the effect of aborting the current task.

`dt_done_when(Handle serverh) ==> integer done_when`
: Indicates the "final progress" for dt_continue(); that is, the value of "progress" that will mean the task is complete (where "progress" is the value dt_continue() returns).

A general algorithm for starting and completing a sort or search task is:

```
Start the task; check the task's return status
If return status is "in progress" then
```

```
        done_when = dt_done_when(server)
        while (status is still "in progress")
            progress = dt_continue(server, status)
            report progress to the user
        endwhile
    endif
```

The following C code fragment illustrates this in more concrete terms:

```
/*** Do the search ***/
progress = dt_mer_similarselect(hitlist, col, searchtype, action, -1,
                          &ret_status, strlen(smiles), smiles, limit,0.0,0.0);
if (ret_status == DX_STATUS_IN_PROGRESS) {
  done_when = dt_done_when(server);
  while (ret_status == DX_STATUS_IN_PROGRESS) {
    printf("Similarity: (%d%%)\n", (100 * progress)/done_when);
    progress = dt_continue(server, &ret_status);
  }
}

/*** Let user know how it came out ***/
if (ret_status == DX_STATUS_NOT_FOUND)
  printf("Target not found - hitlist unchanged\n");
else if (ret_status != DX_STATUS_DONE) {
  printf("Error with similarity search:\n");
  printerrors(stdout, 0);
}
else
  printf("Done: %d hits in list\n", dt_mer_length(hitlist));
```

18.3 Querying for Capabilities

Merlin's set of capabilities includes several ways to sort data, search data, and otherwise examine and modify hitlists. All of Merlin's capabilities are enumerated in the "include" files that come with the Toolkit; however, it is some times desirable to design a user interface without "hard coding" this information. That is, it might be desirable to ask the Toolkit at "run time" for its capabilities, and build the user interface (menus, etc.) using the reported capabilities.

The Merlin Toolkit provides functions that allow you to ask the Merlin system for its capabilities. For example, the sorting function take a "sort type" parameter that indicates how the data are to be sorted (e.g. ASCII, numeric, etc.); using the "capabilities functions" described below, you can ask the server how many types of sorts are available, ask for the "name" of each one, and ask which of these are appropriate for the particular column being sorted. Using the information returned, the program can present this information as a menu from which the user can select the sort-type desired.

The following "capability querying" functions are available:

```
dt_mer_action2name(Handle server, integer action)
dt_mer_function2name(Handle server, integer func)
dt_mer_search2name(Handle server, integer search)
dt_mer_similar2name(Handle server, integer similar)
dt_mer_sort2name(Handle server, integer sort)
dt_mer_subselect2name(Handle server, integer subselect)
dt_mer_superselect2name(Handle server, integer superselect)
```

Each of the above functions returns a string containing a an English- language name for the specified capability. If the capability is unknown (the parameter is out of range) or server is not a server object, returns the invalid string.

```
dt_mer_nactions(Handle serverh)
dt_mer_nfunctions(Handle serverh)
dt_mer_nsearches(Handle serverh)
dt_mer_nsimilars(Handle serverh)
dt_mer_nsorts(Handle serverh)
dt_mer_nsubselects(Handle serverh)
dt_mer_nsuperselects(Handle serverh)
```

Each of the above functions returns an integer equal to the number of valid capabilities. If the capability is unknown, or server is not a server object, returns -1.

The following C code fragment illustrates how one might use these functions to print a list of all legal sorts:

```
nsorts = dt_mer_nsorts(server);
for (sort = 0; sort > nsorts; sort = sort + 1) {
   sort_name = dt_mer_sort2name(&alen,server,sort);
   fprintf(stdout, "%d. %.*s\n", sort, alen, sort_name);
}
```

Two other capability-querying functions are used to help users select capabilities that are appropriate for particular data:

```
dt_mer_sortapplies(Handle column, integer sort) ==> boolean applies
```

Returns TRUE if the specified sort can be applied to the specified column. Sorting methods and the function dt_mer_sort() are discussed below.

```
dt_mer_funcapplies(Handle fieldtype, int func) ==> boolean applies
```

Returns TRUE if the specified function can be applied to create a column of the specified field type. For example, you can't use DX_FUNC_STDDEV (standard deviation) on a fieldtype that is not numeric. Column creation is discussed below.

18.4 Column Objects

A column object is defined by three properties:

Property	**Description**
database	The database (Merlin pool) which is the column's parent object.
fieldtype	Defines which datatype and which field within that datatype is to be used to create the column
function	Describes how to extract the particular field from each TDT.

18.4.1 Column "Functions"

A single THOR datatree (TDT) can contain many occurrences of a particular datatype. For example, a TDT might have dozens or hundreds of names for a compound, such as brand names for a drug. Likewise, it could have many measurements of a particular physical property.

Merlin's "spreadsheet" model (rows and columns) requires that we somehow select from among the multiple occurrences of a particular type of data to create "columns" of data. To do this, we introduce the idea of a column-creation function. These functions provide various methods for choosing among the various occurrences of a particular type of data in a TDT:

`DX_FUNC_FIRST`
: Select the first occurrence of the specified field in the TDT. This is the most commonly used function in column creation.

`DX_FUNC_LAST`
: Select the last occurrence of the specified field in the TDT.

`DX_FUNC_MIN`
: Select the lowest-valued occurrence of the specified field in the TDT. For numbers, this is the lowest numerical value, using a simple "

`DX_FUNC_MAX`
: Select the highest valued occurrence of the specified field in the TDT, as with DX_FUNC_MIN, above, except using ">".

`DX_FUNC_LONGEST`
: Selects the longest (where length is the number of characters in the string).

`DX_FUNC_SHORTEST`
: Selects the shortest.

`DX_FUNC_AVG`
: Creates a column of "derived data" containing the average of all occurrences of the specified fieldtype in the row. If a particular row has no occurrences of the specified fieldtype, the cell in the column will be "not available" (usually indicated by "~"). The fieldtype must be numeric.

`DX_FUNC_STDDEV`
: Creates a column of "derived data" containing the standard deviation of all occurrences of the specified fieldtype in the row. If a particular row has one or zero occurrences of the specified fieldtype, the cell in the column will be "not available". The fieldtype must be numeric.

`DX_FUNC_COUNT`
: Creates a column of "derived data" containing the count of the specified fieldtype in each row. That is, goes through the rows and sums the number of occurrences of the specified fieldtype; the resulting sums become the column's contents.

`DX_FUNC_ALL`
: Creates a column of "pseudo data" which in effect has all occurrences of the specified field type in it. The column initially appears empty; when a search is performed, the entire row is searched; if a field of the specified type is found that matches the search parameters, that field becomes the cell's value. Note that this makes these columns behave somewhat strangely, since their data changes with each search.

18.4.2 Creating Columns

`dt_mer_alloc_column(Handle pool, Handle ftype, integer func) ==> Handle col`
: Creates a column of data from the database pool, using the datafield specified by ftype and the function func.

`dt_mer_getnitems(Handle pool, Handle type) ==> integer nitems`
: If type is a datatype object, returns the number of dataitems in the pool that have the specified datatype. If type is a fieldtype object, returns the number of datafields in the pool that have the specified field type. (Particular implementations of the Merlin server may not be able to report these two numbers separately. In such cases, the server may report the number of

dataitems when you request the number of datafields.)

18.4.3 Information about Columns

`dt_mer_defaultsort(Handle column) ==> integer sort`
: Returns the index of the "most likely" sort type for the specified column. For numeric columns, returns DX_SORT_NUM; for CAS numbers returns DX_SORT_CAS; for all other sortable columns returns DX_SORT_ASC.

`dt_mer_sortapplies(Handle column, integer sort) ==> boolean applies`
: Returns TRUE if the specified sort (a string object) can be applied to the column. For example, you can't sort a numeric column by length, since length only applies to strings.

`dt_mer_function(Handle column) ==> integer func`
: Returns the function that was used to create the column, or -1 if an error is detected.

18.4.4 Polymorphic Functions on Columns

Most (but not all) columns are a "shared resource" on the server: All clients that use a particular column refer to the same actual data. In installations where particular data are frequently used, it is possible to create "permanent" columns, using dt_hold(), that remain in the server's memory, thereby improving startup performance for client programs. For example, you might want to create a permanent SMILES column, a column of you company's ID number, and a column of a particular physical property that all of you users need.

Note that columns of "derived" data, such as similarity columns and columns with the function DX_FUNC_ALL, can't be shared among clients as their contents change with each search. It is not useful to use dt_hold() on such columns.

`dt_hold(Handle column, string thorpassword) ==> boolean ok`
: Marks the specified column "held", so that it will be retained in the Merlin server's memory even when no clients are using it.

`dt_isheld(Handle column) ==> boolean isheld`
: Returns TRUE if column is marked "hold".

`dt_release(Handle database, string execpassword) ==> boolean ok`
: Marks the specified column "released" (not held), so that it will be removed from the Merlin server's memory when the last client deallocates it.

As mentioned in the chapter on datatype objects, column objects respond to requests about datatype and datafield properties. The following functions work when used with column objects; they are described in more detail in the chapter on Datatype objects:

```
dt_datatype(Handle column) ==> Handle datatype
dt_fieldtype(Handle column) ==> Handle fieldtype
dt_dfnorm(Handle obj, int norm) ==> boolean isnorm
dt_dfnormdata(Handle obj, int norm) ==> string normdata
dt_name(Handle obj) ==> string name
dt_briefname(Handle obj) ==> string briefname
dt_summary(Handle obj) ==> string summary
dt_tag(Handle obj) ==> string tag
dt_description(Handle obj) ==> string description
```

18.5 Hitlist Objects

A hitlist object represents an ordered set of rows from a Merlin database. Client programs typically have one primary hitlist that is used for search and sort operations, and often have auxiliary hitlists for "save/restore" and "undo" operations.

While it is possible to create as many hitlists as you like, you should remember that each one uses memory in the server (4 bytes per row in the pool). In general you should use as few as will suffice for the task at hand.

Rows in a hitlist are identified by their index in the hitlist, typically referred to as "ihit" (index of the hit).

18.5.1 Creating Hitlists

`dt_mer_alloc_hitlist(Handle database) ==> Handle hitlist`
: Creates a hitlist. The hitlist is initially "reset" -- it contains all rows in the pool in "native" order.

18.5.2 Retrieving Data: Cells

`dt_mer_cellvalue(Handle column, Handle hitlist, integer ihit) ==>` string cell
: Returns the value of the "cell" -- the value from the `ihit` position of the hitlist in the specified column. The string returned should be used or copied immediately; the Toolkit may reuse the buffer that this function returns on the next call to the Merlin Toolkit.

`dt_mer_getdata(Handle hitlist, int ihit, Handle ftype, int n) ==> string data`
: Allows you to retrieve data without first creating a column: returns the nth occurrence of a specific fieldtype in row ihit of hitlist. The parameter `ftype` is a fieldtype object, and indicates what type of data is desired. It allows you to retrieve data (e.g. SMILES, conformation) whether or not you have a column of that type.

18.6 Sorting

To sort data in Merlin, one specifies a hitlist/column pair, thus defining the cells whose data are to be sorted, along with a sort method. The sort method specifies how the cells are to be compared to one another to determine which is "lowest" and which is "highest". There is a variety of sort methods available, as follows:

`DX_SORT_ASC`
: Sort the data using straight ASCII comparison. Note that in the ASCII character set, all uppercase letters [A-Z] are less than all lowercase letters [a-z], so "Baker" will come before "able". If one string is a prefix of another, the longer string is considered to be greater than the shorter; thus "able-bodied" would come after "able".

`DX_SORT_ANC`
: "Sort, no case" -- sort using straight ASCII comparison, but all lowercase characters [a-z] are converted to their equivalent uppercase [A-Z] before the comparison is made, thus eliminating case distinction. For example, "able" would come before "Baker".

`DX_SORT_ANW`
: "Sort, no whitespace" -- sort using straight ASCII comparison, but ignore "whitespace" characters (space, tab, newline, and carriage- return -- ASCII 32, 7, 10, and 13, respectively). That is, it is equivalent to first removing all whitespace from the strings, then sorting by DX_SORT_ASC.

`DX_SORT_ANP`
: "Sort, no punctuation" -- sort using straight ASCII comparison, but ignore punctuation characters. Punctuation characters are anything that is not alphanumeric ([A-Z], [a-z], and [0-9]). That is, it is equivalent to first removing all punctuation from the strings, then sorting by DX_SORT_ASC.

`DX_SORT_ANCP -- "Sort, no case, no punctuation"`
`DX_SORT_ANCW -- "Sort, no case, no whitespace"`
`DX_SORT_ANPW -- "Sort, no punctuation, no whitespace"`
`DX_SORT_ANCPW -- "Sort, no case, no punctuation, no whitespace"`
: Each of these is a combination of sorts discussed previously.

`DX_SORT_AAZ`
: "Sort, ASCII A-Z only" -- sort ignoring all characters except a-z and A-Z, and ignore case distinction. That is, it is equivalent to removing all non-alphabetic characters from the strings and converting all uppercase characters to their lowercase equivalent, then sorting the data by DX_SORT_ASC.

`DX_SORT_NUM`
: "Sort numerically" -- sort a column of numbers into ascending order.

`DX_SORT_NAB`
: "Sort numerically by absolute value" -- sort a column of numbers into ascending order by magnitude (ignore the sign of the numbers).

`DX_SORT_CAS`
: "Sort CAS numbers" -- sort Chemical Abstracts numbers into ascending order.

`DX_SORT_MFM`
: "Sort by molecular formula" -- sorts molecular formula. Compares each element/number combination as a single "token", so that "C20" is greater than "C2N" (an ASCII sort would put the digit "0" before the letter "N").

`DX_SORT_LEN`
: "Sort by length" -- sorts ASCII data by length; short strings are "lower" than long strings.

One function in the Merlin Toolkit handles all sort methods:

`dt_mer_sort(Handle hitlist, Handle column, integer sortmethod,integer direction, RETURN integer status) ==> progress`

Begins a "sort task" (see Tasks - "Time Slicing", above). Those rows currently in hitlist are sorted using the cells from column. The data are sorted into ascending or descending order according to whether direction is `DX_SORT_ASCENDING` or `DX_SORT_DESCENDING`.

The status of the sort-task is returned in the parameter `status`. The function's return value is either its progress on the task (see dt_done_when()), or -2 if an error is detected. If the hitlist is short enough that the server can sort it in one time-slice, the value of status will be `DX_STATUS_DONE`, and no task will be in progress on the server. Otherwise, the status will be `DX_STATUS_IN_PROGRESS`, and dt_continue() is required to finish the task.

The Merlin server will attempt to choose the most efficient sort technique for the data in the specified column. Whatever method is chosen, the following will be true:

◊ The sort is stable. That is, if two cells are equal, the sort won't affect their relative positions in the hitlist.

◊ The worst-case time it takes to sort a hitlist will grow as N*log(N) time, where N is the length of the hitlist. In some cases it is much faster than this.

`dt_mer_defaultsort(Handle column) ==> integer sortmethod`
: Returns the default sort method for the specified column. The "default" is simply the "most likely" sort a user might choose; there is no real significance to the value this function returns. Returns -1 if column is not a sortable datatype, or if it isn't a column object.

`dt_mer_sortapplies(Handle column, integer sortmethod) ==> boolean` applies
: Returns TRUE if the column can be sorted with the specified sort type, and FALSE if not or if the specified object is not a column object.

18.7 Searching

The Merlin system's most powerful feature is its ability to search a database in a variety of ways. There are five different searching functions in the Merlin Toolkit, to perform string, numeric, similarity, sub- and superstructure searches.

In spite of the variety of searches available, all of the searching functions share most of their parameters; they all look something like the following prototype. We will describe these common parameters here is this pseudo-function definition, and for each actual function only describe those parameters that are unique.

```
dt_mer_xxxsearch(Handle         hitlist,
                 Handle         column,
                 integer        searchtype,
                 integer        action,
                 integer        find_next,
                 RETURN integer status,
                 ...other parameters) ==> integer progress
```

`hitlist`
: Where the "hits" (the rows that meet the search criteria) will be placed. Depending on the parameter action, it may also determine which rows are searched.

`column`
: The data that are to be searched. In some cases, such as a similarity search or a column created with DX_FUNC_ALL, the column's contents also change as a result of the search.

`searchtype`
: Most searches have "submodes" -- for example, when searching for strings, one can choose to ignore case, whitespace, and/or punctuation.

`action`
: Specifies what is to be done with the search results. This is discussed in detail in the following subsection, Actions.

`find_next`
: If action is one of `DX_ACTION_NEXT_HIT` or `DX_ACTION_NEXT_NONHIT`, this parameter specifies where in the hitlist the search is to begin. The search begins at the hit after this value; to search from the hitlist's beginning, specify -1. To continue searching from a previously-found hit, specify that hit's index (the value returned by the previous invocation of the search function).

`status`
: All searches become "tasks" on the server (see the section entitled Tasks -- "Time Slicing", above). This return parameter indicates the status of the search task. If it is `DX_STATUS_IN_PROGRESS`, the search is not complete, and dt_continue() is required to continue the task. If it is `DX_STATUS_DONE`, the search is complete and the function's return value is the hitlist's length, or for "find-next" actions, the hit's index in the hitlist. If it is

DX_STATUS_NOTFOUND, the search is complete but failed to find anything; the hitlist is unchanged. If it is DX_STATUS_ERROR, an error was detected; the task is complete, and the hitlist is unchanged.

progress
: The return parameter for all search functions is their progress on the task. If status is DX_STATUS_IN_PROGRESS, then the function dt_done_when() will return a number which, when divided into progress, yields the fraction of the task that is completed. If status is DX_STATUS_DONE, then progress is the hitlist's new length. If status is DX_STATUS_NOT_FOUND, progress is not defined. If status is DX_STATUS_ERROR, the progress is -2.

18.7.1 Actions

The rows that are to be searched in the pool, and how the results of a search are to be combined with the original hitlist, are defined by an action. There are seven possible actions:

DX_ACTION_NEW_LIST
: The original hitlist is discarded (cleared). All rows in the database are searched; all rows that meet the search criteria are added to the hitlist.

DX_ACTION_ADD_HITS
: All rows not on the original hitlist are searched; rows that meet the search criteria are added to the end of the hitlist.

DX_ACTION_ADD_NONHITS
: All rows not on the original hitlist are searched; rows that don't meet the search criteria are added to the end of the hitlist.

DX_ACTION_DEL_HITS
: The rows in the original hitlist are searched; rows the meet the search criteria are removed from the hitlist.

DX_ACTION_DEL_NONHITS
: The rows in the original hitlist are searched; rows that do not meet the search criteria are removed from the hitlist.

DX_ACTION_NEXT_HIT
: The rows in the original hitlist are searched; as soon as a row is found that meets the search criteria, its hitlist index is returned. The hitlist is unchanged. The data in derived-data columns, such as similarity and columns created using DX_FUNC_ALL, will be altered by the search even though the hitlist is unchanged. The parameter find_next to the search functions (described above) indicates where in the hitlist the search is to begin: The first row examined is find_next + 1.

DX_ACTION_NEXT_NONHIT
: Like DX_ACTION_NEXT_HIT, but returns the first row that does not match the search criteria.

18.7.2 Parametric Searches

There are two types of parametric searches: string and numeric. Only one or the other applies, according to whether the column is a numeric type or not (see dt_dfnorm()).

The parameters `hitlist`, `column`, `action`, `find_next`, `status`, and the return value `progress` are described in the description of the pseudo-search function `dt_mer_xxxsearch()`, above.

```
dt_mer_strsearch(Handle hitlist,
                 Handle  column
                 integer searchtype,
                 integer action,
                 integer find_next,
                 RETURN  integer status,
                 string  s1,
                 string  s2) ==> integer progress
```

Searches the specified column for string-based values `s1` and/or `s2` according to the parameter `searchtype` as detailed in the dt_mer_strsearch() manual page.

```
dt_mer_numsearch(Handle  hitlist
                 Handle  column
                 integer action,
                 integer find_next,
                 RETURN  integer ret_status,
                 float   low_limit,
                 float   high_limit) ==> integer progress
```

Searches the specified column for all numbers in the range `low_limit` to `high_limit`, inclusive. There is no separate "exact match" search for numbers; for this case search with the two limits equal.

Note that that unlike the other search functions, this function has no `searchtype` parameter; there is only one type of numeric search.

18.7.3 Structural Searches

There are three types of structural searches in the Merlin Toolkit: similarity, substructure and superstructure. All structural searches typically make use of "outside" data -- data not in the specified column -- in that they implicitly use the fingerprint data (datatype FP) in the database. If fingerprints are not available, structural searches will work much more slowly.

```
dt_mer_similarselect(Handle  hitlist
               Handle  column
               integer similartype,
               integer action,
               integer find_next,
               RETURN  integer ret_status,
               string  smiles,
               float   limit,
               float   alpha,
               float   beta) ==> integer progress
```

Searches for structures similar to the structure specified by the given SMILES string. Similarity searches are unusual in that the column you specify is a derived-data column: the similarity for each row is computed and stored in the column, then compared to `limit` to determine if the structure meets the search criteria. Substructure searches also make implicit use of the "Fingerprint" (FP) datatype to compute the similarity values; if a particular row doesn't have a fingerprint, its similarity will be "not available".

The parameters `hitlist`, `action`, `find_next`, `status`, and the return value `progress` are described above in the description of the pseudo-search-function `dt_mer_xxxsearch()`, above. The parameter `column` is as described in `dt_mer_xxxsearch()`, but additionally it must be a column of the pseudo-datatype `SIMILARITY`. The parameter `similartype` can be either `DX_SIMILAR_TANIMOTO` or `DX_SIMILAR_EUCLIDIAN`

```
dt_mer_subselect(Handle  hitlist,
                 Handle  column,
                 integer searchtype,
                 integer action,
                 integer find_next,
                 RETURN  integer status,
                 string  smiles) ==> integer progress
```

Searches for substructures of `smiles`. The parameter searchtype is essentially "reserved" for future use -- `DX_SUBSTRUCT_SMILES` is presently the only allowed value.

The parameters `hitlist`, `column`, `action`, `find_next`, `status`, and the return value `progress` are described above in the description of the pseudo-search-function `dt_mer_xxxsearch()`, above.

```
dt_mer_superselect(Handle hitlist,
                   Handle  column,
                   integer searchtype,
                   integer action,
                   integer find_next,
                   RETURN  integer ret_status,
                   string  smiles) ==> integer progress
```

Searches for superstructures of smiles. The interpretation of `smiles` depends on the `searchtype` parameter, as follows:

`search_type == DX_SUPER_SMILES`
: The parameter smiles is interpreted as a SMILES string. Using SMILES, one can specify "ordinary" substructures -- substructures that have exactly-specified atoms and bonds (i.e. no SMARTS expressions).

`search_type == DX_SUPER_SMARTS`
: The parameter `smiles` is interpreted as a SMARTS string. Using SMARTS, one can specify substructures that have expressions for atoms and bonds.

`search_type == DX_SUPER_SMILESPART`
: The parameter smiles is interpreted as a SMILES string. Using SMILES, one can specify "ordinary" substructures -- substructures that have exactly-specified atoms and bonds (i.e. no SMARTS expressions). This search type uses the special FPP<> dataitem, if available, for screening. This search is used to rapidly find substructures within dot-separated components of SMILES, typically applicable for databases of mixtures.

`search_type == DX_SUPER_SMARTSPART`
: The parameter `smiles` is interpreted as a SMARTS string. Using SMARTS, one can specify substructures that have expressions for atoms and bonds. This search type uses the special FPP<> dataitem, if available, for screening. This search is used to rapidly find substructures within dot-separated components of SMILES, typically applicable for databases of mixtures.

During a SMILES and SMARTS searches, implicit use is made of the "Fingerprint" datatype for screening purposes. If fingerprints are not available, the search may be considerably slower.

18.7.4 Program-Object searches The Merlin server's searching capabilities can be extended via the use of user-written *program objects*.

The general topic is discussed in the chapter on program objects. "Attaching" a program object to a merlin server is discussed in the merlinserver manual page. For a specific example of program objects, see the "contrib" directory:

```
$DY_ROOT/contrib/src/progob/merlinbintalk.c
```

A Merlin server can have several program objects attached to it. They are referenced by index, which by convention in this manual we call `iprogob`. The following two functions tell you how many program objects there are and their names:

`dt_mer_nprogobs(Handle server) ==> Integer N`

Reports the number of program objects attached to the Merlin server.

`dt_mer_progob2name(Handle server, Integer iprogob) ==> String name`

Gets the name of the program object attached to the Merlin server. The parameter `iprogob` indicates which program object, and ranges from 0 to N-1, where N is the number of program objects reported by dt_mer_nprogobs(), above.

Currently, Merlin program objects work strictly with "binary" data, such as fingerprints, and produce floating-point results, in a "Similarity" column. There are two tasks each Merlin program object can perform:

- ◊ Given a string (e.g. a SMILES) and some parameters, generate binary data from that string (e.g. a fingerprint).
- ◊ Given binary data (e.g. a fingerprint), compare it to every row in a column of binary data and return a result (e.g. similarity).

More specifically, the following two functions perform these tasks:

```
dt_mer_progob_compute(Handle server,
               Integer iprogob,
               Handle string_object,
               Handle parameters) ==> Handle string
```

Sends the contents of "string_object", followed by the contents of "parameters" (a sequence of string objects), to the program object attached to "server" indicated by "iprogob", and returns a string object containing binary data computed by the program object. The binary data are ASCII encoded; see dt_binary2ascii() for details.

```
dt_mer_progob_compare(Handle  hitlist,
                      Handle  target_column,
                      Handle  result_column,
                      Integer iprogob,
                      Handle  pattern_binary,
                      Handle  parameters,
                      RETURN integer status) ==> progress
```

Uses a program object (see dt_mer_progob_compute(3)) to compare a binary datafield to the contents of a column of binary data, and stores the results of the comparisons in a column of numeric data. The binary data are ASCII encoded; see dt_binary2ascii() for details.

Program objects may require additional parameters to direct their computations and comparisons. For example, a fingerprinting program's computation function might take parameters controlling the size of the fingerprint and the maximum pathlength to follow in generating the fingerprint; a program object that compares mass spectra might take parameters controlling the relative importance of increasing mass. The above two functions both have parameter named, amazingly enough, "parameters". These are sequence-of-strings objects that can take any arbitrary parameters that you need to pass to the program objects. The parameters must be represented in string form (e.g. numeric parameters must be represented in printed ASCII characters). The interpretation of these parameters is strictly up to the program object; the Merlin server simply forwards them to the program object without interpretation.

The program objects can also supply titles for these parameters. Two functions are provided for this purpose:

```
dt_mer_progob_computeparams(dt_Handle server, dt_Integer iprogob)
==> Handle seq_of_strings
```

Asks a program object, via the Merlin server, to report the names and default values for the parameters used by the function dt_mer_progob_compute(..., parameters).

`dt_mer_progob_compareparams(dt_Handle server, dt_Integer iprogob) ==> Handle seq_of_strings`

Asks a program object, via the Merlin server, to report the names and default values for the parameters used by the function dt_mer_progob_compare(..., parameters).

18.8 Other Hitlist Operations

`dt_mer_clear(Handle hitlist) ==> boolean ok`

Clears a hitlist. Returns the number of hits in the hitlist (i.e. zero), or -2 if an error is detected.

`dt_mer_combinehitlists(Handle h1, Handle h2, int action) => integer` nhits

Combines two hitlists in the same manner as the search operations. That is, `h1` is treated as the original hitlist, `h2` is treated as the result of a search; the two hitlists are combined using `action`, and the result placed in `h1`.

`dt_mer_hit2id(Handle hitlist, int ihit) ==> integer id`

There are two ways of identifying a row in a pool:

hit index:

(Called "ihit") An index into a hitlist. This index is used for all operations on Merlin hitlists.

row id

(Called "id") An arbitrary but unique integer that identifies a particular row. You can ask for the id of a row in a hitlist, then use its id to find its position in another hitlist or in a modified version of the original hitlist. An id has no other use. An id is guaranteed to be invariant and unique over the life of a pool object.

Converts a row's hitlist index to its id. Returns the id, or -2 if an error is detected. A typical use of the id is to find a row's id, perform a search or sort, then convert the id back to `ihit`, the row's index in the modified hitlist. See dt_mer_id2hit(), below.

`dt_mer_id2hit(Handle hitlist, int id) ==> integer ihit`

Converts a row's "id" to its hitlist index, "ihit" (see dt_mer_hit2id(), above). Returns the row's index ("ihit"), or -2 if an error is detected. It is an error if the specified row is not in the hitlist.

`dt_mer_invert(Handle hitlist) ==> boolean ok`

Inverts the hitlist: All hit rows become non-hits and all non-hit rows become hits. The current order is lost; the new hits are in native order. Returns the number of hits in the resulting hitlist, or -2 if an error is detected.

`dt_mer_length(Handle hitlist) ==> integer length`

Returns the number of hits in a hitlist.

`dt_mer_mvbottom(Handle hitlist, int ihit) ==> integer nhits`

Moves the specified hit to the end of the hitlist. Returns the (unaltered) hitlist length, or -2 if an error is detected.

`dt_mer_mvtop(Handle hitlist, int ihit) ==> integer nhits`

Moves the specified hit to the top of the hitlist. Returns the (unaltered) hitlist length, or -2 if an error is detected.

`dt_mer_native(Handle hitlist) ==> integer nhits`

Reorders the hitlist (without changing its contents) to "native" order. Returns the number of hits (which is unchanged), or -2 if an error is detected.

"Native" order is essentially arbitrary. It sometimes corresponds to the order in which data are loaded into a database, but it should not be assumed that this is the case. The only thing guaranteed about "native" order is that it won't change during the life of the parent database object (dt_parent(hitlist)).

`dt_mer_reset(Handle hitlist) ==> integer nhits`

Resets and reorders a hitlist so that all rows in the pool are in it in "native" order. Returns the number of hits in the hitlist, or -2 if an error is detected.

`dt_mer_reverse(Handle hitlist) ==> integer nhits`

Reverses the order of the hitlist, without changing its contents. Returns the (unaltered) hitlist length, or -2 if an error is detected.

`dt_mer_zapabove(Handle hitlist, integer ihit) ==> integer nhits`

Deletes all hits above (but not including) the specified hitlist index `ihit`. If `ihit` is greater than or equal to the number of hits, all hits are deleted. If `ihit` is zero or less, no hits are deleted. Returns the hitlist's new length, or -2 if an error is detected.

`dt_mer_zapbelow(Handle hitlist, integer ihit) ==> integer nhits`

Deletes all hits below (but not including) the specified hitlist index `ihit`. If `ihit` is greater than the number of hits, no hits are deleted. If `ihit` is less than zero, all hits are deleted. Returns the hitlist's new length, or -2 if an error is detected.

`dt_mer_zapna(Handle hitlist, Handle column) => integer nhits`

Deletes rows from hitlist for which there is no data in column. Returns the hitlist's new length, or -2 if an error is detected.

`dt_mer_zapnonunique(Handle hitlist, Handle column) ==> integer nhits`

Deletes all rows from hitlist which, in `column`, have the same value as the previous row in the hitlist. Returns the hitlist's new length, or -2 if an error is detected.

18.9 Saving and Restoring Hitlists

It is often necessary to save a hitlist so that it may be restored for later use, or shared with other users. For example, one might be interested in a particular subset of a database; one could use Merlin's searching capabilities to make a hitlist consisting of that subset, then save it.

The Daylight system has no concept of "indices" or other arbitrary identifiers that might be used to save a hitlist. One must use identifiers such as SMILES as the contents of a saved hitlist. Users and programmers should be aware of the implications of this: If a particular row has no identifier loaded in the Merlin Pool, it can't be saved in a hitlist -- there is no way to name it.

Hitlists are stored as TDT files. Each TDT is a "minature" version that contains only the identifier's tag and the identifier. For example, the following hitlist might be from a database that has a number of SMILES-rooted TDT, and a number of entries for which we have no structure, only a company ID number:

```
$SMI<Oc1ccccc1>|
$SMI<OCc1ccccc1>|
$CID<234-54A>|
$SMI<Oc1c(O)cccc1>|
$CIC<235-55B>|
```

Each row in a Merlin pool has a "root identifier" -- the root of the TDT for that row. If a pool's rows are "split out" into subtree rows (i.e. the subtree identifiers have the _P in their datatype definition), their root idntifier will be the subtree's root identifier. These root identifiers are used to store hitlists:

`dt_mer_getroot(Handle hitlist, integer index) => string id`

Returns the root identifier for the specified row, as a TDT string containing only the root identifier's tag and the root identifier (e.g. "$TAG<ID>|").

`dt_mer_sethits(Handle hitlist, Handle column, Handle sos) ==> integer nset`

Sets hits in a hitlist, using a sequence of identifiers and a column whose datatype is that of the identifiers being restored.

The parameter `sos` is a sequence of string objects, each string-object of which should contain a "$TAG<ID>|" string as described in dt_mer_getroot(), above, or a SMILES string (if the first character of the string is not a "$", then it is assumed to be a SMILES; otherwise it is assumed to be a TDT). Each identifier is added to the hitlist. Note that the hitlist is NOT cleared before the additions begin; long lists of identifiers can be added by breaking them into smaller groups. For efficiency, such groups should not be too small, say several hundred to several thousand identifiers at a time.

Returns the number of hits in the sequence that were actually set. This can be different from the sequence's length if one or more identifiers can't be found in the column, or if one or more identifiers is duplicated, or was already in the hitlist. Returns -1 on error.

Note: As of release 4.33, only columns with datatype SMILES ($SMI) and the pseudo-datatype "RowID" ($ROWID) will work with this function. $ROWID columns are by far the most useful, since you can feed them a sequence-of-strings object with mixed datatypes, i.e. one you got from dt_mer_getroot(), above.

19. Widgets

19.1 Introduction

The Daylight Widget Toolkit (tm) provides powerful set of functions for displaying chemical information in a X-windows environment. Each "widget" is designed to handle a particular task that is needed in typical programs for chemical-information processing The widgets in the Widget Toolkit are:

- ◊ **Depict Widget:** Displays depictions and, optionally allows a user to select one or more of them. The Depict Widget is by far the most complex of the Daylight Widgets, as it has several modes of operation.
- ◊ **3D Widget:** Displays conformations, allows the users to rotate the depictions with simple mouse action using a "trackball" conceptual model.
- ◊ **TDT Widget:** Displays THOR Datatrees, provides editing capabilities so users can enter and modify chemical information in a THOR database. Has several display options that allow different levels of detail to be shown.
- ◊ **Edgar Widget:** Graphic-attribute editor. Allows users to change the colors and styles of graphics, such as depictions and conformations, and to save the attributes in a personal options file.
- ◊ **File Widget:** Allows the user to select a file. While finding the file of interest, users can browse through the file system, selectively show only files of the desired type or all files, order the display alphabetically or by name, and cancel the selection.
- ◊ **Status Widget:** A simple widget that graphically shows the progress of a long operations, and provides a timeout and interactive-abort mechanisms.
- ◊ **Message Widget:** Displays errors and warnings returned by the Daylight Toolkit's error functions. Allows users to scroll back through a series of errors and to clear the error display.

The philosophy behind the design of the Daylight Widgets is quite different from other Daylight Toolkits. Because the X Window programming environment is really only useable by C programs, Daylight Widgets are only callable by C programs -- there is no FORTRAN or Pascal interface.

Certain programming constructs that don't work well in mixed-language environments, such as pointers to functions, are not used in the other Daylight Toolkits, but the Widget Toolkit makes heavy use of them.

In addition, there is no concept of polymorphism or objects in the Widgets Toolkit. Each widget is referred to by an identifier, but it is not a "Handle" in the sense used by the other Toolkits.

19.2 Widget Functional Interface

The Daylight Widgets Toolkit shares a key idea with the regular Daylight Toolkits: both use a functional interface. That is, the Daylight Widget Toolkit has no externally-visible C structs, "common blocks", or the like. When a widget is created, it is associated with an id, a simple integer. This identifier is then used for all further operations on the widget: the widget is invoked, hidden, displayed, and destroyed by passing this identifier to widget functions.

By design, the internal operation of all Daylight Widgets is hidden from the application program that uses the widgets. For example, the Daylight GRINS widget is a very sophisticated widget that provides users with a powerful tool for specifying molecules and depictions, but from the application program's point of view, it is simply a source of depiction objects. That is, the operation of GRINS is hidden; all the application has to do is invoke the GRINS widget and wait for a depiction to come back.

The basis of the functional interface is the idea that a widget provides a "service" or "functionality" that is independent of the particular implementation. For example, the GRINS widget task is to return a depiction object to the calling program. How it does that is irrelevant to the calling program: a "cheap imitation" GRINS widget might ask the use to enter a SMILES then use the Daylight SMILES and Depict Toolkits to generate a depiction. This cheap imitation GRINS widget would meet the functional specification, and the application program would be happy (the users, of course, would be unhappy; the real GRINS widget is a powerful graphical input tool for molecular structure).

As another example of the functional nature of the Widgets, consider the Edgar Widget. When invoked, it provides users with the ability to change the graphics (colors, line styles) being used throughout a program, and to save changes in an options file. A user can, for example, change the colors being used by the Depict Widget, change the "depth cue" colors used by the 3D widget, then save these changes for use the next time the program is invoked. Yet with all of these capabilities, the application program has essentially no interaction with the Edgar widget other than to invoke it.

We might summarize this idea by saying the programmer's view of widgets has nothing to do with their operation, only their function.

19.3 Widget Callback Functions

Modern user interfaces, such as Motif and XView, use an event-driven architecture in which input from users (keyboard and mouse activities) drive the application program. Specific actions, such as opening a database, drawing a picture, or starting a database search, are "bound" to particular keys and buttons; these actions are invoked by the event mechanism when the user presses the key or mouse button.

To operate in this environment, Daylight Widgets use "callback" functions to return their results. For example, the File Widget allows users to peruse the file system and select a file; when the user is done the selected file is returned to the application program by invoking an application-program-supplied function that was given to the File Widget for this purpose.

19.4 Options

The Daylight Widgets make use of the Daylight "option manager". The option manager is not a separate program (as the name might suggest), but rather is an integral part of the Daylight Toolkit. It is not used by the other Daylight Toolkit products (except to verify licenses), but is heavily used by Daylight Widgets.

Options specify the configurable behavior of widgets. They typically include the initial location of each widget, the colors that are used for depictions, conformations, and so on, and fonts.

The Daylight Installation Guide contains more information about the options manager. The specific options used by each widget are described on the manual page for each widget.

19.5 The Widgets

Below is an outline of each widget's capabilities. As the saying goes, "A picture is woth a thousand words." In this case, it is difficult to express in words what is obvious to the eye. Most readers of this document will have some aquaintance with the actual widgets, so we will not attempt to describe the widget's operation in detail.

All widgets have a set of "standard" functions; most also have functions specific to the widget. Note that the standard functions are not polymorphic like regular Daylight Toolkit functions; there are actually separate functions for each widget, but they follow a standard naming convention. Each widgth has the following, where "xxx" is replaced by the widget's name:

`int dw_xxx_create(Frame parent);`
: Creates a widget as a child of XView Frame parent. The widget is initially invisible and has nothing in it. A positive widget id is returned on success, 0 is returned on failure; this id is used by all other widget functions. In general, any number of each type of widget can exist at one time, but in most situations one of each type of widget is appropriate.

`void dw_xxx_destroy(int id);`
: Destroys the widget .

`void dw_xxx_hide(int id);`
: Makes the widget invisible.

`void dw_xxx_redraw(int id);`
: Causes the widget to redraw its contents.

`void dw_xxx_reset(int id);`
: Clears the visual and logical contents of the widget. May also reset the widget's label to its default; see the specific description of each widget.

`void dw_xxx_setlabel(int id, char *label);`
: Changes the widget's title.

`void dw_xxx_show(int id);`
: Makes the widget visible. This function is not typically needed unlsee dw_xxx_hide() is used, as the dw_xxx_invoke() functions automatically make the widget visible.

The following is a brief description of each widget.

19.5.1 3D or "Trackball" Widget

The "Trackball" widget, so called because of the visual model used, provides the ability to display a single conformation. The widget can be thought of as "write-only" -- you pass it a conformation and it does everything else; there is no callback function or return value.

19.5.2 Depict Widget

The depict widget is the most complex of the Daylight Widgets, as it has several modes of operation and has a callback function to return results in several forms.

The typical sequence of operations on a Depict Widget are:

- ◊ Create the widget
- ◊ Add depictions to the widget. One or more objects, either depictions or sequence objects that contain depictions, can be added.
- ◊ Invoke the widget using one of the modes described below. This makes it visible and causes the depictions it contains to be displayed.

The Depict Widget's modes are:

- ◊ **depict mode:** Depictions are displayed only.
- ◊ **pickone mode:** Depictions are displayed; the user can select one depiction in a "thoughtful" manner, pressing an "OK" button when the selection is made. The depiction is returned via a callback function.
- ◊ **pickhot mode:** Identical to the pickone mode except that the depiction is selected in a "hasty" manner: the selection is made immediately when the depiction is selected rather than via an "OK" button.
- ◊ **pickmany mode:** Depictions are displayed, and the user can select any number (including zero) of the depictions. The "OK" button causes a sequence of depiction to be returned via the callback function.

19.5.3 Edgar Widget

The Edgar Widget (Edit Graphical Attribute Resources) is a color- picker utility which allows users to asociate real properties (such as color, line style, and width) with functionally-defined graphic attributes (such as background, border, bonds, and atom labels).

From the application program's point of view, the Edgar Widget is simply invoked; the Edgar Widget returns no values.

19.5.4 File Widget

The file widget allows users to peruse the file system and select a file for some particular operation. Its specification allows the application program great flexibility in selecting those files that are to be shown to the user; for example, users could be shown only writeable files with a particular suffix.

The File Widget has two callback functions. The first is called when the user selects a file; the file selected is passed to this callback function. The second callback function is a "filter" -- each file is passed to this function before being shown to the user; the function returns 1 or 0 according to whether the file is to be shown to the user or not.

19.5.5 Font Utility

(Not a widget.) The font utility, dw_font(), is a utility that accesses the Daylight Options manager, and creates an XView menu showing the optional fonts. It is provided as a convenience, and is used internally by some of the Daylight Widgets.

19.5.6 GRINS Widget

The GRINS Widget is a powerful editor that allows fast specification of molecular structure using a graphical molecule editor. GRINS was designed specifically for input tasks; it is not intended to replace molecule editors that produce publication-quality pictures. Instead, GRINS is optimized for fast entry of structure, and includes features such as templates and "parent" molecules that speed this task.

The GRINS widget has one callback function, that is invoked with a depiction as its parameter when the user selects "OK".

19.5.7 Help Widget

The Help Widget provides online user manuals, and allows context- dependent positioning in the manuals. Help-Widget files are simply 80-character-wide text files containing section headings that serve as the keys to context-sensitive invocation.

The Help Widget is invoked with a topic and key. For example, invoking it with the topic "thor" and the key "CREATING DATABASES" would cause it to find the file "thor.hw", then look for the string "CREATING DATABASES". The widget would be made visible with "thor.hw" as its contents, positioned at the key of interest.

19.5.8 Message Widget

The Message Widget is a convenient was of allowing the user to browse the message sent to the error facility provided by the Daylight Toolkit. It has no callback functions.

19.5.9 Status Widget

The Status Widget keeps the user informed of the progress of lengthy operations. It shows a graphical "thermometer" display that fills up as the task proceeds, and shows a textual summary of the task's progress, including an estimated time of completion.

In addition, the Status Widget has a timeout and abort facility. The application program can indicate a time at which the process will be interrupted; at this point the user can choose to proceed, set another timeout, or abort the operation. In addition, the user can at any point abort the operation.

The Status Widget is somewhat more difficult to use than the other widgets due to the design of XView; a special main event loop must be written for XViews event processing. The Widget Toolkit comes with an example illustrating how this is done.

19.5.10 TDT Widget

The TDT Widget provides for the display and editing of THOR Datatrees. It uses two callback functions:

◊ A "done" callback function, which is invoked when the user is finished editing the TDT and presses "Save".
◊ A "datatypes" callback, which should return a sequence of the datatypes that are to be available to the person editing the datatree.

If the TDT supplied is from a read-only database, then the TDT widget automatically uses the "Browse" mode; no editing is possible. Otherwise, the widget has add, delete, move, and modify modes that allow data to be entered and modified.

19.6 Widget Programmer's Reference

The details of calling syntax, features, and functionality for programming the Widgets Toolkit are described in a separate document, a set of UNIX-style " man pages". It is called the Daylight Widgets Toolkit Reference.

20. HTTP Toolkit

Overview: This is the comprehensive reference for Daylight's HTTP Toolkit and is for use by programmers. Covered topics include: 1) the Application Programmers Interface, 2) server & CGI program structure, 3) the essential properties used to construct programs, 4) the entire set of properties, and 5) example routines and program source code demonstrating use of the toolkit. **What is the HTTP Toolkit?** The HTTP Toolkit is a web-based programmer library containing a well-behaved set of entry points and conforms to specifications of RFC 1945 "*Hypertext Transfer Protocol -- HTTP/1.0*" (available at http://www.ietf.org/rfc/rfc1945.txt). The HTTP Toolkit allows you to receive requests to and send responses from Daylight tools over the web. The HTTP Toolkit is a faster, simplier alternative to Common Gateway Interface (CGI) programming and Apache modules. **Why the HTTP Toolkit?** The HTTP Toolkit was created because most people have a browser, such as Internet Explorer or Netscape Navigator, so the client application is already installed! The HTTP Toolkit leverages the ubiquitity of the internet by enabling programmers to serve information over the web. **Prerequisites** The intended audience for this manual is programmers. A modest knowledge of UNIX and the C programming language are key ingredients for understanding this manual. Familiarity with UNIX commands and the ability to configure the UNIX environment is a must. Experience in C programming is necessary because examples are illustrated using the C language. Further, knowledge of HTTP headers and HTML syntax is helpful.

Daylight Software. You will need to download and install the software from http://www.daylight.com/download/index.html and a license for the SMILES and HTTP Toolkits (for licensing, email info@daylight.com). Source code for programs listed as figures in this manual are available in the "Contrib" area of your Daylight installation ($DY_ROOT/contrib/src subdirectory).

Web Server Configuration. To use a HTTP Toolkit program as a CGI, your local web server will need the LD_LIBRARY_PATH and DY_LICENSEDATA environment variables as well as Daylight's Installation Guide. **Organization** The HTTP Toolkit is not difficult to understand, but you must understand the Application Programmer Interface (API) and how properties control behavior before you start programming.

Section 20.1. *The Application Programmer Interface*, defines the programmer library entry points, arguments, and behavior.

Section 20.2. *The Structure of Programs*, illustrates how to use the API to create "run-once" and "run-forever" programs.

Section 20.3. *The Essential Properties*, shows the key ingredients of receiving a request and sending a response and illustrates the classic example, "Hello World!".

Section 20.4. *The Entire Set of Properties*, lists a description of all properties and illustrates use of POST data in the example, "Canonical SMILES".

Section 20.5. *Working with the Toolkit*, illustrates use of the toolkit with programming examples.
Conventions This manual uses the following typographical conventions:

Italics
: Used for URLs, filenames, variables, new terms where they are defined, and emphasis

`Constant width`
: property names, routine names, source code, computer output, and any literal text

`Constant width bold`
: Used in examples to show commands or test that would be typed literally by you

ALL CAPS
: Used for object names, environment variables and HTML tags **Scope**

Although the intended use of this toolkit is for deployment of Daylight tools, the HTTP Toolkit is a general web server; any kind of information can be deployed using this tool.

20.1 The Application Programmer Interface

The Application Programmer Interface (API) implements the following objects:

HTTP Toolkit Object Classes	
HTTP	server object for encapsulation of the HyperText Transfer Protocol
TRANSFER	message transfer object for handling requests & responses

and consists of the following entry points:

◊ dt_alloc_http_server
◊ dt_http_get
◊ dt_http_put

That's it! The `dt_alloc_http_server` entry point is used to create a service on the web. The `dt_http_get` and `dt_http_put` entry points are used to receive and respond to requests, respectively.

20.1.1 Create a Service (dt_alloc_http_server)

The entry point for creating a web service is:

dt_Handle dt_alloc_http_server(dt_Integer *port*);

This allocates a new HTTP object. The *port* argument is a number in the range of 0 to 65535 and identifies the process on the network. A port number of 0 means that the service behaves as a CGI. In other words, it runs *once* and terminates. A port number greater than zero means the service listens on that port and behaves as a server. In other words, it runs *forever*. The return value of this routine is the handle of a new HTTP object or NULL_OB if an error is detected. See the manual page on dt_alloc_http_server for more information.

20.1.2. Receive a Request (dt_http_get)

The entry point for receiving a request is:

dt_Handle dt_http_get(dt_Handle *http*);

This allocates a new TRANSFER object. The *http* argument is the HTTP object returned from `dt_alloc_http_server`. The return value is the handle of a new TRANSFER object or NULL_OB if no request is received or an error is detected. See the manual page on dt_http_get for more information.

20.1.3 Send a Response (dt_http_put)

The entry point for sending a response is:

dt_Boolean dt_http_put(dt_Handle *xfer*);

This sends a response. The *xfer* argument is the TRANSFER object returned from `dt_http_get`. The return value is TRUE if data is successfully transmitted, or FALSE if an error is detected. See the manual page on dt_http_put for more information.

20.2 The Structure of Programs

A HTTP Toolkit program can be designed to function as a CGI service (run *once*) or a web service (run *forever*). Also, a program can be designed to function as *either* one or the other. The following sections contain source code written in the C language for each of the three programs.

The following examples test the API and default behavior of the Toolkit. The TRANSFER object, as returned from `dt_http_get`, contain properties reflecting the request. Most noteworthy is the Uniform Resource Locator (URL) property. Since the following examples do nothing with request properties and the toolkit initializes the status code to 404 (meaning the URL is not found), we expect each call to `dt_http_put` to produce a "`404 Not Found`" response. This behavior is a valid HTTP service, although not useful beyond testing. Nonetheless, it is encouraged that you follow details and reproduce results, so you can proceed with confidence into more complicated aspects of the Toolkit.

20.2.1 Build a CGI

The following source code is an example of a main routine designed to function as a CGI:

Figure 20.2.1-1. Program Source Code: CGI

```
#include "dt_smiles.h"
#include "dt_http.h"

int main() {
  dt_Handle  http, xfer;
  dt_Boolean stat;
```

```
 6. ....
      /* create a service */
 7. ....
      http = dt_alloc_http_server(0);
 8. ....
 9. ....  /* receive a request */
10. ....  xfer = dt_http_get(http);
11. ....
      /* send a response */
12. ....
      stat = dt_http_put(xfer);
13. ....
14. ....  return !stat;
15. }....
16. ....
17. ....
18. ....
```

This is a *bare bones* program that creates a service, receives a request, and sends a response. The Daylight include files are order-depenent (lines 1 & 2). The port argument (line 9) is 0 indicating a CGI service. Nothing is done with the request (line 12), so the response (line 15) is the default response ("`404 Not Found`"). The program is missing error checking and memory deallocation, but that's okay, the Toolkit can handle it. Let's make a note to tidy it up later and move ahead for now.

This program serves as a fundamental test of HTTP Toolkit CGI service. To test CGI service, save the above code as `test-http-cgi.c` and make the program (or you can get and make it from the "Contrib" area of Daylight Software).

To compile on Red Hat Linux or SGI Irix systems:

```
cc -o test-http-cgi test-http-cgi.c -I$DY_ROOT/include -L$DY_ROOT/lib -ldt_http -ldt_smi
```

On Sun Solaris systems:

```
cc -o test-http-cgi test-http-cgi.c -I$DY_ROOT/include -L$DY_ROOT/lib -ldt_http -ldt_smi
```

On Macintosh Darwin (OSX) systems:

```
cc -o test-http-cgi test-http-cgi.c -I$DY_ROOT/include -L$DY_ROOT/lib -ldt_http -ldt_smi
/System/Library/Frameworks/IOKit.framework/Versions/A/IOKit
/System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation
EOF
```

Now we've got our HTTP Toolkit CGI application compiled, let's use it. Execute the program with end-of-file markers:

```
./test-http-cgi << EOF
GET / HTTP/1.0
EOF
```

You should see output HTML code that contains the following:

```
404 Not Found

The server cannot find the requested URL.

Daylight/4.81 Server at www.daylight.com Port 0
```

The program received the request and responded "`404 Not Found`". In your output, the version (`4.81`) and host machine (`www.daylight.com`) will vary and additional header lines (`Date`, `Server`, and `Content`) may appear. If you're not sure if your program output is correct, compare it to test-http-cgi.ref (see "Contrib" area of Daylight Software), which is an example of complete output from the CGI program.

Before we move on, let revisit the CGI program source code and tidy it up:

Figure 20.2.1-2. Program Source Code: CGI (revised)

```
 1. #include <stdio.h>
 2. #include "dt_smiles.h"
 3. #include "dt_http.h"
 4.
 5. int main() {
 6.   dt_Handle  http, xfer;
 7.   dt_Boolean stat = 1;
 8.
 9.   /* create a service */
10.   if (NULL_OB == (http = dt_alloc_http_server(0))) {
11.     fprintf(stderr, "dt_alloc_http_server failed\n");
12.     return 1;
13.   }
14.
15.   /* receive a request */
16.   if (NULL_OB != (xfer = dt_http_get(http))) {
17.     /* send a response */
18.     stat = dt_http_put(xfer);
19.     dt_dealloc(xfer);
20.   }
21.   dt_dealloc(http);
22.
23.   return !stat;
24. }
```

Compared to the original CGI program, the main portion (Figure 20.2.1-1, lines 9, 12 and 15) are logically reworked (lines 10 through 18) to handle errors and memory deallocation. Now, let's move on to a server application.

20.2.2. Build a Server

The following source code is an example of a main routine designed to function as a server.

Figure 20.2.2 Program Source Code: Server

```
#include <stdio.h>
#include "dt_smiles.h"
#include "dt_http.h"

int main(int argc, char **argv) {
  dt_Handle  http, xfer;
  dt_Integer port = 0;

  if ((2 != argc) || (1 != sscanf(argv[1], "%d", &port))) {
    fprintf(stderr, "%s: missing integer argument\n", argv[0]);
    return 1;
  }
  if ((0 >= port) || (65535 < port)) {
    fprintf(stderr, "%s: port number out-of-range (1-65535)\n", argv[0]);
    return 1;
  }

  /* create a service */
  if (NULL_OB == (http = dt_alloc_http_server(port))) {
    fprintf(stderr, "dt_alloc_http_server failed\n");
    return 1;
  }

  for (;;) {
    /* receive a request */
    xfer = dt_http_get(http);

    if (NULL_OB != xfer) {
      /* send a response */
      dt_http_put(xfer);
      dt_dealloc(xfer);
    }
  }
}
```

The essential difference between this code and the previous CGI example is that the calls to `dt_http_get` and `dt_http_put` are within an infinite loop.

This program serves as a fundamental test of HTTP Toolkit web service. To test the web service, save the above code as `test-http-server.c` and make the program (see the previous section, or the "Contrib" area of Daylight Software):

To compile on Red Hat Linux or SGI Irix systems:

```
cc -o test-http-server test-http-server.c -I$DY_ROOT/include -L$DY_ROOT/lib -ldt_http -l
```

On Sun Solaris systems:

```
cc -o test-http-server test-http-server.c -I$DY_ROOT/include -L$DY_ROOT/lib -ldt_http -l
```

On Macintosh Darwin (OSX) systems:

```
cc -o test-http-server test-http-server.c -I$DY_ROOT/include -L$DY_ROOT/lib -ldt_http -l
/System/Library/Frameworks/IOKit.framework/Versions/A/IOKit
/System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation
EOF
```

Then, execute the program:

```
./test-http-server 1234
```

This runs the web service on port 1234. Point your browser to the machine running the service and place a ":1234" at the end of the location, e.g., *http://www.daylight.com:1234*. This web service produces the same output as the CGI, except for the port number.

```
Not Found

The requested URL / was not found on this server.

Daylight/4.81 Server at www.daylight.com Port 1234
```

In your output, the version (`4.81`) and host machine (`www.daylight.com`) will vary and additional header lines (`Date`, `Server`, and `Content`) may appear. If you're not sure if your program output is correct, compare it to test-http-server.ref (see "Contrib" area of Daylight Software), which is an example of complete output from the server program.

20.2.3. Build a Dual-Purpose Server & CGI

The following source code is an example of a main routine designed to function as a server or a CGI.

Figure 20.2.3 Program Source Code: Dual-Purpose Server & CGI

```
#include <stdio.h>
#include "dt_smiles.h"
#include "dt_http.h"

int main(int argc, char **argv) {
  dt_Handle  http, xfer;
  dt_Integer port = 0;
  dt_Boolean stat = 1;

  if ((2 == argc) && (1 != sscanf(argv[1], "%d", &port))) {
    fprintf(stderr, "%s: missing integer argument\n", argv[0]);
    return 1;
  }
  if ((0 > port) || (65535 < port)) {
    fprintf(stderr, "%s: port number out-of-range (0-65535)\n", argv[0]);
    return 1;
  }

  /* create a service */
  if (NULL_OB == (http = dt_alloc_http_server(port))) {
    fprintf(stderr, "dt_alloc_http_server failed\n");
    return 1;
  }

  for (;;) {
    /* receive a request */
    xfer = dt_http_get(http);

    if (NULL_OB != xfer) {
      /* send a response */
```

```
            stat = dt_http_put(xfer);
            dt_dealloc(xfer);
        }
        /* run-once for CGI */
        if (0 == port)
            break;
    }

    dt_dealloc(http);
    return !stat;
}
```

The essential difference between this code and the previous example is allowing port 0, breaking out of the loop, and memory deallocation.

This program serves as a fundamental test of HTTP Toolkit web service and CGI service in one program. To test the dual-purpose service, save the above code as `test-http-server-cgi.c` and make the program (or see "Contrib" area of Daylight Software).

To compile on Red Hat Linux or SGI Irix systems:

```
cc -o test-http-server-cgi test-http-server-cgi.c -I$DY_ROOT/include -L$DY_ROOT/lib -ldt_
```

On Sun Solaris systems:

```
cc -o test-http-server-cgi test-http-server-cgi.c -I$DY_ROOT/include -L$DY_ROOT/lib -ldt_
```

On Macintosh Darwin (OSX) systems:

```
cc -o test-http-server-cgi test-http-server-cgi.c -I$DY_ROOT/include -L$DY_ROOT/lib -ldt_
/System/Library/Frameworks/IOKit.framework/Versions/A/IOKit
/System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation
EOF
```

Now, you can execute the program as a CGI:

```
./test-http-server-cgi << EOF
GET / HTTP/1.0
EOF
```

and as a server:

```
./test-http-server-cgi 1234
```

You should get the same results as the CGI and server programs in Section 20.2.1 and Section 20.2.2. If you've reproduced these results, you're on your way towards deployment of Daylight tools via a web sevice. Now, let's get into the essential properties of receiving requests and sending responses.

20.3 The Essential Properties

In the Daylight Toolkit, "named properties" associate name/value pairs with objects. In the HTTP Toolkit, named properties are used to get request and set response information associated with the TRANSFER object. For example, the TRANSFER object has a named property called `htt_path` reflecting the value of the requested URL. Let's suppose the URL is *http://www.daylight.com/dayhtml/*. If so, the TRANSFER object's named property `htt_path` (a property name) and would be "/dayhtml/" (its value).

20.3.1 Get a Request

The following lists the name, description, and datatype for the essential properties associated with receiving requests:

Table 20.3.1. Essential Properties of a Request

Name	Description	Datatype
htt_method	request method	dt_String
htt_path	URL or path	dt_String
htt_protocol	protocol and version	dt_String

◊ **htt_method** is the request method, for example, "GET", "HEAD", or "POST". Other values include "PUT", "DELETE", "LINK", and "UNLINK".
◊ **htt_path** is the URL or path, for example, *http://www.daylight.com* or */*.
◊ **htt_protocol** is the protocol and version, for example, "HTTP/1.0" or "HTTP/1.1".

To get the essential request properties:

Figure 20.3.1. Example Source Code: Get a Request Property

```
method   = dt_string(&mlen, xfer, 10, "htt_method");
path     = dt_string(&plen, xfer,  8, "htt_path");
protocol = dt_string(&rlen, xfer, 12, "htt_protocol");
```

The `htt_method_id`, `htt_path`, and `htt_protocol` are property names and `method`, `path`, and `protocol` contain the property values. For more information, see the manual page for `dt_string`.

20.3.2 Set a Response

The following table lists the name, description, and datatype for the essential properties associated with sending responses.

Table 20.3.2. Essential Properties of a Response

Property Name	Description	Datatype
htt_code	status code (200, 404, etc.)	dt_Integer

htt_mime_type	mime type (text/html, image/gif, etc.)	dt_String
htt_content	program data (HTML, GIF image, etc.)	dt_String

◊ **htt_code** is the response status code and is initialized to 404 (Not Found) when a TRANSFER object is returned from `dt_http_get`. This property should be set to 200 for response that are "OK".
◊ **htt_mime_type** is the MIME type of the content, for example "text/html" or "image/gif", and is initialized to "text/html" when a TRANSFER object is returned from `dt_http_get`. The syntax conforms to RFC 1521 "*MIME (Multipurpose Internet Mail Extensions)*" avavilable at *http://www.ietf.org/rfc/rfc1520.txt*. When this property is not NULL, the value is used with the "Content-Type" header field.
◊ **htt_content** is the content of the response with NULL as the initial value. This property is the body of the response sent from the server to the client. When this property is not NULL, the length in bytes of the content is used with the "Content-Length" header field.

To set the essential response properties:

Figure 20.3.2. Example Source Code: Set a Response Property

```
....
dt_setinteger   (xfer,  8, "htt_code",      200);
dt_setstring    (xfer, 13, "htt_mime_type",   9, "text/html");
dt_setstring    (xfer, 11, "htt_content",    12, "Hello World!");
dt_appendstring (xfer, 11, "htt_content",    12, "\n\nMethod:   ");
dt_appendstring (xfer, 11, "htt_content", mlen, method);
dt_appendstring (xfer, 11, "htt_content",    11, "\nPath:     ");
dt_appendstring (xfer, 11, "htt_content", plen, path);
dt_appendstring (xfer, 11, "htt_content",    11, "\nProtocol: ");
dt_appendstring (xfer, 11, "htt_content", rlen, protocol);
dt_appendstring (xfer, 11, "htt_content",     1, "\n");
....
```

For more information, see the manual page for `dt_setinteger`, `dt_setstring` and `dt_appendstring`.

20.3.3 Processing the GET Method: Hello World!

Now we have the essential ingredients to build our first program that gets request and sets response properties. Now, let's work through a program for the GET method. Consider the following "Hello World!" program:

Figure 20.3.3. Program Source Code: Hello World!

```
#include <stdio.h>
#include "dt_smiles.h"
#include "dt_http.h"
....
static void my_handler(dt_Handle xfer);
....
int main(int argc, char **argv) {
  dt_Handle  http, xfer;
  dt_Integer port = 0;
  dt_Boolean stat = 1;
....
```

```
  if ((2 == argc) && (1 != sscanf(argv[1], "%d", &port))) {
    fprintf(stderr, "%s: missing integer argument\n", argv[0]);
    return 1;
  }
  if ((0 > port) || (65535 < port)) {
    fprintf(stderr, "%s: port number out-of-range (0-65535)\n", argv[0]);
    return 1;
  }

  /* create a service */
  if (NULL_OB == (http = dt_alloc_http_server(port))) {
    fprintf(stderr, "dt_alloc_http_server failed\n");
    return 1;
  }

  for (;;) {
    /* receive a request */
    xfer = dt_http_get(http);

    if (NULL_OB != xfer) {
      /* process request, prepare response */
      my_handler(xfer);
      /* send a response */
      stat = dt_http_put(xfer);
      dt_dealloc(xfer);
    }
    /* run-once for CGI */
    if (0 == port)
      break;
  }

  dt_dealloc(http);
  return !stat;
}

static void my_handler(dt_Handle xfer) {
  dt_String method, path, protocol;
  dt_Integer mlen, plen, rlen;

  method   = dt_string(&mlen, xfer, 10, "htt_method");
  path     = dt_string(&plen, xfer,  8, "htt_path");
  protocol = dt_string(&rlen, xfer, 12, "htt_protocol");

  dt_setinteger   (xfer,  8, "htt_code",      200);
  dt_setstring    (xfer, 13, "htt_mime_type",   9, "text/html");
  dt_setstring    (xfer, 11, "htt_content",    12, "Hello World!");
  dt_appendstring (xfer, 11, "htt_content",    12, "\n\nMethod:   ");
  dt_appendstring (xfer, 11, "htt_content", mlen, method);
  dt_appendstring (xfer, 11, "htt_content",    11, "\nPath:     ");
  dt_appendstring (xfer, 11, "htt_content", plen, path);
  dt_appendstring (xfer, 11, "htt_content",    11, "\nProtocol: ");
  dt_appendstring (xfer, 11, "htt_content", rlen, protocol);
  dt_appendstring (xfer, 11, "htt_content",     1, "\n");
}
```

```
....
....
....
....
....
....
```

The only difference between this code and the previous "Dual-Purpose Server & CGI" example is the `htt_handler` routine, which gets the method, path, and protocol request properties then sets the return code, mime type, and data response properties.

This program serves as a test of HTTP Toolkit essential properties. To test the "Hello World!" service, save the above code as `http-hello-world.c` and make the program (or see "Contrib" area of Daylight Software).

To compile on Red Hat Linux or SGI Irix systems:

```
cc -o http-hello-world http-hello-world.c -I$DY_ROOT/include -L$DY_ROOT/lib -ldt_http -l
```

On Sun Solaris systems:

```
cc -o http-hello-world http-hello-world.c -I$DY_ROOT/include -L$DY_ROOT/lib -ldt_http -l
```

On Macintosh Darwin (OSX) systems:

```
cc -o http-hello-world http-hello-world.c -I$DY_ROOT/include -L$DY_ROOT/lib -ldt_http -l
/System/Library/Frameworks/IOKit.framework/Versions/A/IOKit
/System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation
EOF
```

Execute the program as a CGI:

```
./http-hello-world << EOF
GET / HTTP/1.0
EOF
```

You should see output HTML code:

```
Hello World!

Method:   GET
Path:     /
Protocol: HTTP/1.0
```

The program received the request, got the URL path and responded "`Hello World!`" with the request's properties appended to it. If you're not sure if your program output is correct, compare it to http-hello-world.ref (see "Contrib" area of Daylight Software), which contains the complete output from the program.

We've covered the essential properties and worked through a program for the GET method. Now, let's cover the entire set of HTTP Toolkit properties.

20.4. The Entire Set of Properties

Properties are organized by two categories:

◊ Read-Only
◊ Read-Write

Each category is grouped into two subcategories:

◊ CGI/Server
◊ Request/Response

CGI/Server properties are associated to the HTTP object. Request/Response properties are associated with the TRANSFER object.

20.4.1. Read-Only Properties of a CGI/Server

Each of the following properties are set when the HTTP server is allocated and do not change, except for `htt_status`, which changes with each request. The following table lists the named property, description, and datatype for the read-only properties associated with a CGI/Server.

Table 20.4.1. Read-Only Properties of a CGI/Server

Name	Description	Datatype
htt_date	HTTP object allocation date & time	dt_String
htt_status	service state	dt_Integer
htt_port	communication endpoint number	dt_Integer
htt_host	local host name	dt_String
htt_lips	local host IP address	dt_String

◊ **htt_date** is the date & time of HTTP object allocation, for example, Mon, 02 Dec 2002 07:40:03 GMT. The format conforms to RFC 822 "*Standard for the Format of ARPA Internet Text Messages*" and RPC 1123 "*Requirements for Internet Hosts -- Application and Support*" and avavilable at *http://www.ietf.org/rfc/rfc822.txt* and *http://www.ietf.org/rfc/rfc1123.txt*.
◊ **htt_status** is the state of the HTTP service when a TRANSFER object is returned from `dt_http_get` and one of following values:

DX_HTT_OK - a request has been received. The TRANSFER object will be equal to NULL_OB if the request was processed internally. Otherwise, the TRANSFER object will be a valid object for processing.

DX_HTT_TIMEOUT - a request has not been received. The TRANSFER object will be equal to NULL_OB.

DX_HTT_ERROR - an error condition exists. The TRANSFER object will be equal to NULL_OB. Use `dt_errors` to access error information.

DX_HTT_DONE - the same as DX_HTT_OK and also indicates that the service has completed. This status is useful for breaking a CGI service from a dual-purpose Server/CGI loop. Calling `dt_http_get` again will result in an error.

◊ **htt_port** is the port number on which the service is listening. This value is equal to the argument passed to `dt_alloc_http_server`. Valid values are 0 to 65535 inclusive. Port 0 is interpreted as standard input and causes the service to behave as a CGI.

◊ **htt_host** is the name of the local host as viewed from the server using the `uname` system call, for example, *www.daylight.com*.

◊ **htt_lips** is the IP address of the local host as viewed from the server using `gethostbyaddr` system call, for example, *207.225.60.130*.

20.4.2 Read-Write Properties of a CGI/Server

Each of the following properties can be set anytime *after* a HTTP server object is allocated and affects the behavior of `dt_http_get`. The following table lists the named property, description, datatype, and default value for the read-write properties associated with a CGI/Server.

Table 20.4.2. Read-Write Properties of a CGI/Server

Name	**Description**	**Datatype**	**Default**
htt_access_log	enable logging of processed requests	dt_Boolean	FALSE
htt_method_id	request method identifiers allowed	dt_Integer	DX_HTT_METHOD_ALL
htt_timeout_ms	time to listen for a request	dt_Integer	5000

◊ **htt_access_log** is a boolean to indicate logging of processed requests. When TRUE, successful responses will be written to the error queue at the DX_ERR_NOTE level. Applications should process the queue per request. Valid values for this propery are TRUE and FALSE.

◊ **htt_method_id** is a bitwise OR of request method identifiers that are implemented by the service. If a request method received within a call to `dt_http_get` is not implemented, the toolkit will internally process the request with a "`501 Not Implemented`" response. This is useful for denying unwanted methods within the toolkit. Valid values for this property are:

DX_HTT_METHOD_HEAD - the service implements the request method HEAD.

DX_HTT_METHOD_GET - the service implements the request method GET.

DX_HTT_METHOD_POST - the service implements the request method POST.

DX_HTT_METHOD_ALL - a bitwise OR of all of the above method identifiers.

DX_HTT_METHOD_ANY - disables toolkit internal processing of unimplemented requests.

◊ **htt_timeout_ms** is the time in milliseconds the service polls the listening port for a request by using the `select` system call. This value determines how long a call to `dt_http_get` will wait before returning NULL_OB and setting the `htt_status` property to DX_HTT_TIMEOUT.

20.4.3 Read-Only Properties of a Request

The following table lists the named property, description, and datatype for the read-only properties associated with a request/response.

Table 20.4.3. Read-Only Properties of a Request

Name	**Description (HTTP header field)**	**Datatype**
htt_agent	software information (User-Agent)	dt_String
htt_authorize	username & password (Authorization)	dt_String
htt_from	user email address (From)	dt_String
htt_method	request method	dt_String
htt_method_id	method identifier	dt_Integer
htt_path	URL or path	dt_String
htt_post	POST field & value pairs	dt_Handle
htt_protocol	protocol and version	dt_String
htt_rawhead	original HEAD data	dt_String
htt_rawpost	original POST data	dt_String
htt_recv_	prefix for additional headers	dt_String
htt_referer	referral URL (Referer)	dt_String
htt_rips	remote host IP address	dt_String
htt_since	date condition (If-Modified-Since)	dt_String

◊ **htt_agent** is the information about the software used to make the request, for example, "Mozilla/4.78 [en] (X11; U; Linux 2.4.7-10 i686)"
◊ **htt_authorize** is the username & password used to authorize a request.
◊ **htt_from** is the email address of the user.
◊ **htt_method** is the request method, for example, "GET", "HEAD", or "POST". Other values include "PUT", "DELETE", "LINK", and "UNLINK".
◊ **htt_method_id** is the request method identifier of the request. Valid values for this property are:

DX_HTT_METHOD_HEAD - the request method is HEAD.

DX_HTT_METHOD_GET - the request method is GET.

DX_HTT_METHOD_POST - the request method is POST.

DX_HTT_METHOD_NONE - the request method is none of the above implemented methods. In this case, the request method can be determined using the `htt_method` property.
◊ **htt_path** is the URL or path, for example, *http://www.daylight.com* or */*.

◊ **htt_post** contains the request posts field/value pairs stored as a handle to a sequence of string objects. Each string object represents one header line from the request. The header field name is the string value of the string object and the header field value is the value of the "htt_value" property of the string object. See the example on how to get a POST value below.
◊ **htt_protocol** is the protocol and version, for example, "HTTP/1.0" or "HTTP/1.1".
◊ **htt_rawhead** is the original HEAD portion of the data request. This string contains line delimiters (usually \r\n, \n, or \r) and ends with a blank line.
◊ **htt_rawpost** is the original POST portion of the data request. This string is NULL for methods other than POST.
◊ **htt_recv_** is the property name prefix for receiving unimplemented request headers and have no predefined property name. For such headers, this prefix is is concatinated with the header name to construct a property to store the header value. For example, the "Cookie" header is not implemented, so when a "Cookie" header is received, the `htt_recv_Cookie` property will contain the "Cookie" header value. In this way, the toolkit can support any request header information. For information on cookies, see RFC 2965 "*HTTP State Management Mechanism*" at http://www.ietf.org/rfc/rfc2965.txt.
◊ **htt_referer** is the URL of the referring resource.
◊ **htt_rips** is the IP address of the remote host as viewed from the server using the `accept(3socket)` system call. If the service is a CGI, the `REMOTE_ADDR` environment variable is used.
◊ **htt_since** is the conditional date of which the request. The format is the same as the `htt_date` property. If the resource is older than this date, the resource is not requested.

The following source code shows how to get the value of the POST field named "smiles":

Figure 20.4.3. Example Source Code: Get POST Data

```
/* du_http_post - return the string object of a POST named property
**                 return NULL_OB if the name is not found
*/
dt_Handle du_http_post(dt_Handle xfer, dt_Integer plen, dt_String pname) {
  dt_Handle sequence, string;
  dt_String propname;

  /* get property */
  sequence = dt_handle(xfer, 8, "htt_post");

  /* loop over sequence, get strings */
  while (NULL_OB != (string = dt_next(sequence))) {
    /* get propname */
    propname = dt_stringvalue(&plen, string);
    /* check for match */
    if (0 == strncmp(propname, pname, plen))
      /* return string value */
      return string;
  }

  /* not found */
  return NULL_OB;
}
```

For more information, see dt_handle, dt_next, dt_stringvalue, and dt_string.

20.4.4 Read-Write Properties of a Response

The following table lists the named property, description, datatype, and default value for the read-write properties associated with a request/response:

Table 20.4.4. Read-Write Properties of a Response

Name	Description (HTTP header field in parenthesis)	Datatype	Default
htt_authenticate	username & password challenge (WWW-Authenticate)	dt_String	NULL
htt_code	status code	dt_Integer	404
htt_content	content body	dt_String	NULL
htt_date	date & time (Date)	dt_String	NULL
htt_encoding	content compression (Content-Encoding)	dt_String	NULL
htt_expires	content expiration date & time (Expires)	dt_String	NULL
htt_location	URL redirection (Location)	dt_String	NULL
htt_method_id	implemented methods (Allow)	dt_Integer	NULL
htt_mime_type	content MIME type (Content-Type)	dt_String	text/html
htt_modified	content last change date & time (Last-Modified)	dt_String	NULL
htt_pragma	implementation-specific directives (Pragma)	dt_String	NULL
htt_send_	prefix for additional headers	dt_String	NULL
htt_service	service identification (Server)	dt_String	NULL
htt_version	protocol and version	dt_String	HTTP/1.0
HTML Presentation			
htm_autohtml	truthfulness of HTML auto-generation	dt_Integer	FALSE
htm_body_tag	<BODY> tag	dt_String	NULL
htm_body_bg_color	parameter BGCOLOR within <BODY&gt tag	dt_String	NULL
htm_body_bg_image	parameter BACKGROUND within <BODY&gt tag	dt_String	NULL
htm_doctype_tag	<!DOCTYPE> tag	dt_String	NULL
htm_favicon	parameter HREF within <LINK&gt tag	dt_String	NULL
htm_head_tag	<HEAD> tag	dt_String	NULL
htm_head_title	<TITLE> tag within <HEAD&gt tag	dt_String	NULL
htm_head_script	<SCRIPT> tag within <HEAD&gt tag	dt_String	NULL
htm_head_style	<STYLE> tag within <HEAD&gt tag	dt_String	NULL

htm_prefix	prefix to content	dt_String	NULL
htm_postfix	postfix to content	dt_String	NULL

◊ **htt_authenticate** is the challenge for a client to authenticate itself and is used with status code 401. If this property is not set with status code 401, the toolkit will override the response with "`500 Internal Server Error`". When this property is not NULL, the value is used with the "WWW-Authenticate" header field.

◊ **htt_code** is the response status code and is initialized to 404 (Not Found) when a TRANSFER object is returned from `dt_http_get`. This property should be set to 200 for response that are "OK".

◊ **htt_content** is the content of the response with NULL as the initial value. This property is the body of the response sent from the server to the client. When this property is not NULL, the length in bytes of the content is used with the "Content-Length" header field.

◊ **htt_date** is the response date & time, for example, Mon, 02 Dec 2002 07:40:03 GMT. The format conforms to RFC 822 "*Standard for the Format of ARPA Internet Text Messages*" and RPC 1123 "*Requirements for Internet Hosts -- Application and Support*" and avavilable at *http://www.ietf.org/rfc/rfc822.txt* and *http://www.ietf.org/rfc/rfc1123.txt*.

◊ **htt_encoding** is the encoding or compression performed on the content, for example, base64 or gzip. When this property is not NULL, the value is used with the "Content-Encoding" header field.

◊ **htt_expires** is the date & time the resource expires, for example, Mon, 02 Dec 2002 07:40:03 GMT. Like the `htt_date` property, the format conforms to RFC 822 and FPC 1123.. This value is used by caches and proxies to store server responses until it expires. When this property is not NULL, the value is used with the "Expires" header field.

◊ **htt_location** is the absolute URL for redirecting a request for a resource and is used with status codes 301 and 302. If this property is not set with status codes 301 and 302, the toolkit will override the response with "`500 Internal Server Error`". When this property is not NULL, the value is used with the "Location" header field.

◊ **htt_method_id** is a bitwise OR of request method identifiers that are implemented by the service for the response content. When this property is not NULL, the value is used with the "Allow" header field. Valid values for this property are:

DX_HTT_METHOD_HEAD - the service implements the request method HEAD.

DX_HTT_METHOD_GET - the service implements the request method GET.

DX_HTT_METHOD_POST - the service implements the request method POST.

DX_HTT_METHOD_ALL - a bitwise OR of all of the above method identifiers.

DX_HTT_METHOD_ANY - disables sending the "Allow" header field.

◊ **htt_mime_type** is the MIME type of the content, for example "text/html" or "image/gif", and is initialized to "text/html" when a TRANSFER object is returned from `dt_http_get`. The syntax conforms to RFC 1521 "*MIME (Multipurpose Internet Mail Extensions)*" available at *http://www.ietf.org/rfc/rfc1520.txt*. When this property is not NULL, the value is used with the "Content-Type" header field.

◊ **htt_modified** is the date & time the resource was last modified. When this property is not NULL, the value is used with the "Last-Modified" header field.

◊ **htt_pragma** is the response implementation-specific directives that may apply to any recipient along the request/response chain. All pragma directives specify optional behavior from the viewpoint of the protocol; however, some systems may require that behavior be consistent with the directives.

◊ **htt_send_** is the property name prefix for sending unimplemented response headers, and have no predefined property name. For such headers, this prefix should be concatinated with the header name to construct a property to store the header value. For example, the "Set-Cookie" header is not implemented, so when a "Set-Cookie" header should be sent, the programmer should set the `htt_send_Set-Cookie` property to contain the "Set-Cookie" header value. In this way, the toolkit can support any response header information. see RFC 2965 "*HTTP State Management Mechanism*" at http://www.ietf.org/rfc/rfc2965.txt.

◊ **htt_service** is the identifier of the service and is set when an object is returned from `dt_http_get`. This property is a free-form string and is set to the toolkit software version, for example, "Daylight/4.81". When this property is not NULL, the value is used with the "Server" header field.

◊ **htt_version** is the response protocol and version and is set when an object is returned from `dt_http_get`. The value is initialized to match the request protocol and version (`htt_protocol` property). So, HTTP/0.9 requests are sent HTTP/0.9 responses (no headers) and HTTP/1.0 requests are sent HTTP/1.0 responses (with headers). Requests that are received using HTTP/1.1 and higher will result in this property being initialized to "HTTP/1.0" within the call to `dt_http_get`. In order to respond to any other request protocol, the programmer must set this property (for example, to "HTTP/1.1") and add appropiate headers using the `htt_send_` property prefix prior to calling `dt_http_put`.

The following properties pertain to HTML Presentation:

◊ **htm_autohtml** is the HTML auto-generation flag. When this property is zero, all of the following properties are ignored.

◊ **htm_body_tag** is the body tag. When this property is NULL, no BODY tag will be used unless one or both of `htm_background` or `htm_bgcolor` properties are non-NULL. For example:

```
<BODY BACKGROUND=sunlogo-gray.gif BGCOLOR=e0e0e0>
```

◊ **htm_body_bg_color** is the body tag background color parameter. When this property is NULL, no BGCOLOR parameter will be used within a BODY tag unless specified in the `htm_body` property, which overrides this property. For example:

```
e0e0e0
```

◊ **htm_body_bg_image** is the body tag background parameter. When this property is NULL, no BACKGROUND parameter is used within a BODY tag unless specified in the `htm_body` property, which overrides this property. For example:

```
sunlogo-gray.gif
```

◊ **htm_doctype_tag** is the document tag. When this property is NULL, the following value is used:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 1.1//EN">
```

◊ **htm_favicon** is the hypertext reference in the link tag of the header. The reference specifies the location of a custom icon to be used as a thumbnail image in the 'address box window' and 'bookmark listing'. A common value for this parameter is "/favicon.ico". When this parameter is not NULL, a LINK tag within the HEAD field will be output. For example:

```
<LINK REL="shortcut icon" TYPE="image/x-icon"
HREF="/favicon.ico" />
```

◊ **htm_head_tag** is the head tag. When this property is NULL, no HEAD tag is used unless any of `htm_title`, `htm_script`, or `htm_style` properties are non-NULL. For example:

```
<HEAD>
<TITLE>Auto-HTML Format Demo</TITLE>
<SCRIPT LANGUAGE=JavaScript
SRC=/application/x-javascript/loadJavaGrins.js></SCRIPT>
<STYLE TYPE=text/css></STYLE>
</HEAD>
```

◊ **htm_head_title** is the title tag. When this property is NULL, no TITLE tag is used unless specified in the `htm_head` property, which overrides this property. For example:

```
<TITLE>Auto-HTML Format Demo</TITLE>
```

◊ **htm_head_script** is the script tag. When this property is NULL, no SCRIPT tag is used unless specified in the `htm_head` property, which overrides this property. For example:

```
<SCRIPT LANGUAGE=JavaScript
SRC=/application/x-javascript/loadJavaGrins.js></SCRIPT>
```

◊ **htm_head_style** is the style tag. When this property is NULL, no STYLE is used unless specified in the `htm_head` property, which overrides this property. For example:

```
<STYLE TYPE=text/css></STYLE>
```

◊ **htm_prefix** is the content that prefixes the `htt_content` property in the response. This is used to specify a banner at the top of each page. For example:

```
<TABLE BORDER CELLSPACING CELLPADDING=4 WIDTH=100%%><TR>
<TD ALIGN=CENTER WIDTH=100%%><A HREF=/>Auto-HTML Format
Demo</A>
</TABLE>
```

◊ **htm_postfix** is the content that postfixes the `htt_content` property in the response. This is used to specify a banner at the bottom of each page. For example:

```
<TABLE BORDER CELLSPACING CELLPADDING=4 WIDTH=100%%><TR>
<TD ALIGN=CENTER><A HREF=mailto:info@daylight.com>
    info@daylight.com</A>
<TD ALIGN=CENTER VALIGN=MIDDLE WIDTH=100%%>
  <A HREF=http://www.daylight.com>
  Daylight Chemical Information Systems, Inc.</A>
<TD ALIGN=CENTER VALIGN=MIDDLE><I>Daylight<BR>HTTP Toolkit</I>
</TABLE>
```

We've covered all properties of the toolkit. Now, let's start using these properties to build various kinds of programs and control the behavior of the toolkit.

20.5 Working With the Toolkit

We've constructed an example program to respond to GET requests ("Hello World!", Figure 20.3.3f). In order to respond to GET request in general, you would parse the `htt_path` property and set the htt_code, htt_mime_type, and htt_content properties. Let's move onto processing POST requests.

20.5.1 Processing the POST Method

Two kinds of POST data content types are supported by the toolkit:

◊ URL-encoding (application/x-www-form-urlencoding)
◊ multipart media (multipart/form-data)

Both kinds of POST data can be obtained from a HTML form that is created using the following source code:

Figure 20.5.1. Example Source Code: Set a POST form

```
/* du_http_form - set form into response
*/
void du_http_form(dt_Handle xfer, char *name, char *type, char *encoding,
                  char *action, char *button, int vlen, dt_String value) {

  /* begin form */
  dt_appendstring(xfer, 11, "htt_content", 35,
      "<FORM METHOD=POST NAME=form ACTION=");
  /* action */
  dt_appendstring(xfer, 11, "htt_content", strlen(action), action);
  /* encoding */
  dt_appendstring(xfer, 11, "htt_content",  9, " ENCTYPE=");
  dt_appendstring(xfer, 11, "htt_content", strlen(encoding), encoding);
  dt_appendstring(xfer, 11, "htt_content",  2, ">\n");
  /* label */
  dt_appendstring(xfer, 11, "htt_content", strlen(name), name);
  /* name */
  dt_appendstring(xfer, 11, "htt_content", 14, ": <INPUT NAME=");
  dt_appendstring(xfer, 11, "htt_content", strlen(name), name);
  /* type */
  dt_appendstring(xfer, 11, "htt_content",  6, " TYPE=");
  dt_appendstring(xfer, 11, "htt_content", strlen(type), type);
  /* value */
  dt_appendstring(xfer, 11, "htt_content",  7, " VALUE=");
  if(NULL != value)
    dt_appendstring(xfer, 11, "htt_content", vlen, value);
  dt_appendstring(xfer, 11, "htt_content",  2, ">\n");
  /* button */
  if (NULL != button) {
    dt_appendstring(xfer, 11, "htt_content", 19, "<INPUT TYPE=BUTTON ");
    dt_appendstring(xfer, 11, "htt_content", strlen(button), button);
    dt_appendstring(xfer, 11, "htt_content",  2, ">\n");
  }
  /* end form */
  dt_appendstring(xfer, 11, "htt_content", 40,
      "<INPUT TYPE=SUBMIT NAME=button>\n</FORM>\n");
}
```

```
....
```

This routine accepts several arguments:

◊ *name* is an indentification label.
◊ *type* is the input field type, for example, "TEXT" or "FILE".
◊ *encoding* is the content type, for example, "application/x-www-form-urlencoding" or "multipart/form-data".
◊ *action* is the target URL.
◊ *button* is for specifying an additional button.
◊ *value* is used to initialize the input field.

Each argument is NULL-terminated, except the *value* argument, which is length-terminated with the *vlen* argument.

The `du_http_form` routine, along with the source code from `du_http_post` (Figure 20.4.3) and the Dual-Purpose Server & CGI (Figure 20.2.3) can be reused in other example program. This code is available in the "Contrib" area of Daylight Software as `du_http_main.c`. The `main` routine is renamed to `du_http_main` so the library won't have a execution entry point, which calls `my_handler(`*`xfer`*`)` just like the "Hello World!" program. A header file for prototypes is called du_http.h. Also, a "makefile" is available for use with the system `make` utility. To compile this code for reuse in other programs, make the `libdu.a` library, for example,

```
cd $DY_ROOT/contrib/src
make
```

All programs in this manual can built using `make`, for example,

```
cd $DY_ROOT/contrib/src/http
make http-hello-world
```

This builds the "Hello World!" program from Figure 20.3.3. Now, let's construct an example program that uses POST data and our new link library `libdu.a`.

20.5.1.1 URL-Encoding (application/x-www-form-urlencoded)

Let's construct an example program that uses POST data using a MIME type called "application/x-www-form-urlencoded". Consider the following "Canonical SMILES" program:

Figure 20.5.1.1. Program Source Code: Canonical SMILES

```
#include <stdio.h>
#include "dt_smiles.h"
#include "dt_http.h"
#include "du_http.h"

/* my_handler - get POST data and set response
*/
static void my_handler(dt_Handle xfer) {
  dt_Handle string;
  dt_String value = NULL;
  dt_Integer vlen = 0;
  char *name = "SMILES", *type = "TEXT";
  char *encoding = "application/x-www-form-urlencoding";
  /* set return code and mime type */
  dt_setinteger(xfer, 8, "htt_code", 200);
  dt_setstring (xfer, 8, "htt_mime_type", 9, "text/html");

  /* get POST data */
  if (NULL_OB != (string = du_http_post(xfer, strlen(name), name)))
    value = dt_string(&vlen, string, 9, "htt_value");

  /* write form */
  du_http_form(xfer, name, type, encoding, "/", NULL, vlen, value);

  /* parse smiles and canonicalize */
  if (NULL != value ) {
    dt_Handle molecule;
    dt_String cansmiles = "invalid";
    dt_Integer clen = 7;

    if (NULL_OB != (molecule = dt_smilin(vlen, value)))
      cansmiles = dt_cansmiles(&clen, molecule, 1);
    /* set response */
    dt_appendstring(xfer, 11, "htt_content", 22, "\n<P>Canonical SMILES: ");
    dt_appendstring(xfer, 11, "htt_content", clen, cansmiles);
    dt_appendstring(xfer, 11, "htt_content",  1, "\n");
    dt_dealloc(molecule);
  }
}

/* main - calls dual-purpose server & CGI
*/
int main(int argc, char **argv) {
  return du_http_main(argc,argv);
}
```

The key aspects of the `my_handler` routine are calls to:

◊ `du_http_post` to get a SMILES from POST data
◊ `du_http_form` to set a POST form of type *TEXT* and encoding *application/x-www-form-urlencoding*
◊ `dt_smilin` and `dt_cansmiles` to get the canonical form of the SMILES

We set "SMILES" to be the name of the text input in the form, then used that name to get to POST data. Then we used the `htt_value` property to get the SMILES string. This code is available as http-cansmi.c in the "Contrib" area of Daylight Software and is made with the command:

```
cd $DY_ROOT/contrib/src/http
make http-cansmi
```

Now, let's execute the the program as a CGI:

```
./http-cansmi << EOF
```

```
POST / HTTP/1.0
Content-type: application/x-www-form-urlencoded
Content-length: 30

SMILES=OCC&button=Submit+Query
EOF
```

You should see output HTML code that contains:

```
Canonical SMILES: CCO
```

The program received the request, got the SMILES from POST data, called the SMILES Toolkit to the get the canonical SMILES and responded "`Canonical SMILES: CCO`". If you're not sure if your program output is correct, compare it to http-cansmi.ref (see "Contrib" area of Daylight Software). This is the complete output from the program.

Now, let's construct an example program that uses multipart/form-data POST data.

20.5.1.2 Multipart Data (multipart/form-data)

Let's construct an example program that uses POST data of a MIME type called "multipart/form-data". Consider the following "Upload File" program:

Figure 20.5.1.2. Program Source Code: Upload File

```
#include <stdio.h>
#include "dt_smiles.h"
#include "dt_http.h"
#include "du_http.h"

/* my_handler - get POST data and set response
*/
static void my_handler(dt_Handle xfer) {
  dt_Handle string;
  dt_String value = NULL;
  dt_Integer vlen = 0;
  char *name = "UPLOAD", *type = "FILE";
  char *encoding = "multipart/form-data";
  dt_String filename = NULL, mimetype = NULL;
  dt_Integer flen = 0, mlen = 0;

  /* set return code */
  dt_setinteger(xfer, 8, "htt_code", 200);

  /* get file from POST data */
  if (NULL_OB != (string = du_http_post(xfer, strlen(name), name))) {
    value    = dt_string(&vlen, string,  9, "htt_value");
    filename = dt_string(&flen, string, 12, "htt_filename");
    mimetype = dt_string(&mlen, string, 13, "htt_mime_type");
  }

  /* set MIME type */
  if (NULL != mimetype) {
    dt_setstring (xfer, 8, "htt_mime_type", mlen, mimetype);
    /* set file content into response and return if the MIME type is not HTML */
    if (0 != strncmp("text/html", mimetype, mlen)) {
        dt_appendstring(xfer, 11, "htt_content", vlen, value);
        return;
    }
  }

  /* write form */
  du_http_form(xfer, name, type, encoding, "/", NULL, flen, filename);

  /* filename */
  if (NULL != filename) {
    dt_appendstring(xfer, 11, "htt_content", 15, "\n<P>Filename:  ");
    dt_appendstring(xfer, 11, "htt_content", flen, filename);
  }
  /* MIME type */
  if (NULL != mimetype) {
    dt_appendstring(xfer, 11, "htt_content", 15, "\n<P>MIME type: ");
    dt_appendstring(xfer, 11, "htt_content", mlen, mimetype);
  }
  /* file contents */
  if (NULL != value) {
    dt_appendstring(xfer, 11, "htt_content", 15, "\n<P>Content:   ");
    dt_appendstring(xfer, 11, "htt_content", vlen, value);
  }
  dt_appendstring (xfer, 11, "htt_content",    1, "\n");
}

/* main - calls dual-purpose server & CGI
*/
int main(int argc, char **argv) {
  return du_http_main(argc,argv);
}
```

This program source code is similar to the previous "Canonical SMILES" program; both use `dt_http_main`, `dt_http_post`, and `dt_http_form` from the `libdu.a` library. A key differences are calls from the `my_handler` routines to:

◊ `du_http_post` to get a file from POST data
◊ `du_http_form` to set a POST form of type *FILE* and encoding *multipart/form-data*

This code is available as http-upload.c in the "Contrib" area of Daylight Software and is made with the command:

```
cd $DY_ROOT/contrib/src/http
make http-upload
```

Now, let's execute the the program as a CGI:

```
./http-upload << EOF
POST / HTTP/1.0
Content-type: multipart/form-data; boundary=---------------------------33163136917725
Content-Length: 300

-----------------------------33163136917725
Content-Disposition: form-data; name="UPLOAD"; filename="foo.html"
Content-Type: text/html

bar
-----------------------------33163136917725
Content-Disposition: form-data; name="button"

Submit Query
-----------------------------33163136917725--
EOF
```

You should see output HTML code that contains:

```
Filename:  foo.html
MIME type: text/html
Content:   bar
```

The program received the request, got the filename, MIME type, and content from POST data, then responded with the value of those properties. If you're not sure if your program output is correct, compare it to http-upload.ref (see "Contrib" area of Daylight Software), which contains the complete output from the program.

The arguments to `du_http_form` reflect the difference between `application/x-www-form-urlencoded` and `multipart/form-data` POST data. We set "UPLOAD" to be the name of the file input in the form, then used that name to get POST data and the `htt_value` property to get the content of the file. Further, `multipart/form-data` POST data can have additional properties for an uploaded file, such as `htt_filename` and `htt_mime_type` which define the name of the file and the MIME type of the content. The `my_handler` routine sets the MIME type of the response to the `htt_mime_type` property of the file and sends the form with the file contents only if the MIME type is "text/html". In this way, we can upload any kind of MIME type, such as "image/gif", and get the form along with the file if the file is HTML.

20.5.2. Authenticating Access

The basic form of user authentication is base64-encoding. When a client (browser) attempts unauthorized access for a protected URL, the server sends a "`401 Unauthorized`" response. This response normally invokes the client to "pop-up" a window prompting the user for a username and password. Upon entering a username and password, the client browser encodes the data using the base64 algorithm. Although the algorithm is simple and can easily be decoded with paper and pencil, base64-encoding obfuscates text beyond easy recognition. Then the client sends the request for the protected URL along with the base64-encoding data to the server. In order for the server to validate the username and password, the base64-encoded data must be decoded. For example:

Figure 20.5.2-1. Example Source Code: Base64 Routines

```



/* FUNCTION: du_http_base64dechar - decode a base 64 character to a 6-bit digit
*/
static char du_http_base64dechar(char c) {
  if ((96 <c) && (123 > c))
    return c - 71;  /* a-z */
  else if ((65 <c) && (91 > c))
    return c - 65;  /* A-Z */
  else if ((47 <c) && (58 > c))
    return c + 4;  /* 0-9 */
  else if ('+' == c)
    return 62;
  else if ('/' == c)
    return 63;
  else /* if ('=' == c) */
    return 0;
}

/* FUNCTION: du_http_base64decode - decode an ASCII string using base 64
*/
char* du_http_base64decode(int len, char *base64, char *ascii) {
  int i, j, k;
  char a, b, c, d;

  for(i=j=0; i/*convert base 64 character to 6-bit digit */
    a = du_http_base64dechar(*(base64 + i++));
    b = (i < len) ? du_http_base64dechar(*(base64 + i++)) : 0;
    c = (i < len) ? du_http_base64dechar(*(base64 + i++)) : 0;
    d = (i < len) ? du_http_base64dechar(*(base64 + i++)) : 0;
    /* join four 6-bit digits into 24-bit integer */
    k = (a << 18) + (b << 12) + (c << 6) + d;
    /* decompose 24-bit integer into three 8-bit characters */
    *(ascii + j++) = k >> 16 & 255;
    *(ascii + j++) = k >>  8 & 255;
    *(ascii + j++) = k       & 255;
  }
  ascii[j] = '\0';
  return ascii;
}



```

This source code is part of the `libdu.a` library.

Now, let's construct an example program that uses base64-decoding to authenticate a username and password:

Figure 20.5.2-2. Program Source Code: Basic Authentication

```
#include <stdio.h>
#include "dt_smiles.h"
#include "dt_http.h"
#include "du_http.h"

/* my_handler - basic authentication
```

```
*/
void my_handler(dt_Handle xfer) {
  dt_String value;
  dt_Integer vlen, klen;
  char *key, *ptr = "Basic realm=\"Daylight Contrib\"";

  /* check for file protected with user/passwd mug/coffee */
  if (NULL == (value = dt_string(&vlen, xfer, 13, "htt_authorize"))) {
    dt_setstring(xfer, 16, "htt_authenticate", strlen(ptr), ptr);
    dt_setinteger(xfer, 8, "htt_code", 401);
    return;
  }

  /* check that encoding is base64 ("Basic"), else respond 401 (Authenticate) */
  if (0 != memcmp(value, "Basic ", 6)) {
    dt_setstring(xfer, 16, "htt_authenticate", strlen(ptr), ptr);
    dt_setinteger(xfer, 8, "htt_code", 401);
    return;
  }
  value += 6;
  vlen  -= 6;

  /* allocate memory for key, else respond 500 (Internal Server Error) */
  klen = (vlen*3)/4;
  if(NULL == (key = (char*)malloc(klen*sizeof(char)))) {
    fprintf(stderr, "out-of-memory (malloc(%d) failed)", klen);
    dt_setinteger(xfer, 8, "htt_code", 500);
    return;
  }

  /* decode using base64 */
  du_http_base64decode(vlen, value, key);

  /* authenticate username/pasword */
  if(0 != memcmp(key, "mug:coffee", 10)) {
    /* incorrect username/password, respond 401 (Authenticate) again */
    ptr = "Basic realm=\"Daylight Toolkit\"";
    dt_setstring(xfer, 16, "htt_authenticate", strlen(ptr), ptr);
    dt_setinteger(xfer, 8, "htt_code", 401);
  } else {
    /* correct username/password, respond OK */
    dt_setinteger(xfer, 8, "htt_code", 200);
    dt_appendstring(xfer, 11, "htt_content",   19, "Base64 decoding of ");
    dt_appendstring(xfer, 11, "htt_content", vlen, value);
    dt_appendstring(xfer, 11, "htt_content",    4, " is ");
    dt_appendstring(xfer, 11, "htt_content", klen, key);
    dt_appendstring(xfer, 11, "htt_content",    1, "\n");
  }

  /* deallocate key */
  free(key);
}

/* main - calls entry point for dual-purpose server & CGI
*/
int main(int argc, char **argv) {
  return du_http_main(argc,argv);
}


```

```
56. ....
57. ....
58. ....
59. ....
60. ....
61. ....
62. ....
63. ....
64. ....
```

This code has several features worth mentioning:

◊ Line 14: The client must send the "WWW-Authorize" header, which is stored in the `htt_authorize` property.
◊ Line 21: The authentication form must be "Basic", which means base64-encoding.
◊ Line 38: The server calls the base64-decoding routine (Figure 20.5.2-1).
◊ Line 41: The username and password must match "mug:coffee". The colon (":") character is inserted between the username and password by the client, and so appears in the string that the server decodes.

This code is available as http-base64.c in the "Contrib" area of Daylight Software and made with the command:

```
cd $DY_ROOT/contrib/src/http
make http-base64
```

Now, let's execute the the program as a CGI:

```
./http-base64 << EOF
GET / HTTP/1.0
EOF


Status: 401 Unauthorized
Date: Tue, 31 Dec 2002 04:52:34 GMT
Server: Daylight/4.81
WWW-Authenticate: Basic realm="Daylight Contrib"
```

Here we attempted access and the server responded "`401 Unauthorized`. Included in the response is the `WWW-Authenticate` header, telling the client that the server expects `Basic` (base64) encoding. The `realm` parameter "Daylight Contrib" appears on the window that "pops-up" prompting for the username and password. This is intended to help the user determine which username and password is appropiate. Now, let's simulate that the user entered the username "mug" and password "coffee":

```
./http-base64 << EOF
GET / HTTP/1.0
Authorization: Basic bXVnOmNvZmZlZQ==
EOF


Date: Tue, 31 Dec 2002 05:24:04 GMT
Server: Daylight/4.81
Content-Length: 52
Content-Type: text/html
```

```
Base64 decoding of bXVnOmNvZmZlZQ== is mug:coffee
```

The base64-encoded string "bXVnOmNvZmZlZQ==" decodes to "mug:coffee" and the client is authorized to access the server resources. Using base64-encoding, you can protect resources using a basic form of authentication.

20.5.3 Serving Files from Disk

As with all other Daylight toolkits, the HTTP Toolkit does not support explicit file I/O. The initial thought on server resources is to have all content *built-in*. This has the advantage of independence from access problems and hardware faults, as well as the disadvantage of not being able to update data without updating the server. Also, serving files from disk has the problem of configuration (and security) for data that should (and shouldn't) be served. There's no reasonable solution deserving implementation within the toolkit. Therefore, this section describes a program that serves files from disk that is *built on top of* the toolkit. For exanmple:

Figure 20.5.3-1. Example Source Code: Disk Access Routine

```
/* du_http_month - return month number given 3-letter month (Jan=0, Feb=1, etc.)
*/
int du_http_month(char *str) {
    if      (0 == memcmp(str, "Jan", 3)) return  0;
    else if (0 == memcmp(str, "Feb", 3)) return  1;
    else if (0 == memcmp(str, "Mar", 3)) return  2;
    else if (0 == memcmp(str, "Apr", 3)) return  3;
    else if (0 == memcmp(str, "May", 3)) return  4;
    else if (0 == memcmp(str, "Jun", 3)) return  5;
    else if (0 == memcmp(str, "Jul", 3)) return  6;
    else if (0 == memcmp(str, "Aug", 3)) return  7;
    else if (0 == memcmp(str, "Sep", 3)) return  8;
    else if (0 == memcmp(str, "Oct", 3)) return  9;
    else if (0 == memcmp(str, "Nov", 3)) return 10;
    else                     /* Dec */   return 11;
}

/* du_http_file - disk access routine
*/
dt_Integer du_http_file(dt_Handle xfer) {
  dt_String path, value;
  dt_Integer plen, vlen;
  char buf[1024], dmy[4], month[4], tz[4];
  char *ptr, *dy_root, *docroot, filename[1024];
  int fd, rlen, year, mday, hour, min, sec;
  size_t len;
  ssize_t bytes, tbytes=0;
  struct stat fdstat;
  struct tm gmt, since;

  /* set document root directory */
  if (NULL == (dy_root = getenv("DY_ROOT"))) {
    dt_setstring (xfer, 11, "htt_content", 15, "DY_ROOT not set");
    dt_setinteger(xfer,  8, "htt_code", 500);
    return 500;
  }
  docroot = "/contrib/src/data/http";
  path    = dt_string(&plen, xfer, 8, "htt_path");

  /* check for access outside document root */
  if (NULL != strstr(path, "../")) {
    dt_setstring (xfer, 11, "htt_content", 15, "filename contains ../");
    dt_setinteger(xfer,  8, "htt_code", 404);
    return 404;
  }

  /* check for overflow */
  if (1024 < strlen(dy_root) + strlen(docroot) + plen + 1) {
    dt_setstring (xfer, 11, "htt_content", 17, "filename too long");
    dt_setinteger(xfer,  8, "htt_code", 500);
    return 500;
  }

  /* construct absolute filename (ignore arguments) */
  sprintf(filename, "%s%s%.*s", dy_root, docroot, plen, path);
  if (NULL != (ptr = strchr(filename, '+')))
    *ptr = '\0';

  /* open file or respond "Not Found" */
  if (-1 == (fd = open(filename, O_RDONLY, 0))) {
    dt_setinteger(xfer,  8, "htt_code", 404);
    return 404;
  }

  /* get file statistics */
  if (-1 == (fstat(fd, &fdstat))) {
    len = sprintf(buf, "error signal %d (%s)", errno, strerror(errno));
    dt_setstring(xfer, 11, "htt_content", len, buf);
    dt_setinteger(xfer,  8, "htt_code", 500);
    close(fd);
    return 500;
  }

  /* check for content */
  if (0 == fdstat.st_size) {
    dt_setinteger(xfer,  8, "htt_code", 204);
    close(fd);
    return 204;
  }

  /* get time-and-date of last modification in RFC 1123 format*/
  if (NULL != gmtime_r(&(fdstat.st_mtime), &gmt))
    if ( 0 < (len = strftime(buf, 32, "%a, %d %b %Y %H:%M:%S GMT", &gmt)))
      dt_setstring(xfer, 12, "htt_modified", len, buf);

  /* check for If-Modified-Since */
  if (NULL != (value = dt_string(&vlen, xfer, 9, "htt_since"))) {
    sscanf(value, "%3s, %2d %3s %4d %2d:%2d:%2d %3s",
           dmy, &mday, month, &year, &hour, &min, &sec, tz);
    /* check date for GMT timezone (RFC 1123 conformance) */
    if (0 != memcmp(tz, "GMT", 3)) {
      len = sprintf(buf, "Invalid date format (%.*s)", vlen, value);
      dt_setstring(xfer, 11, "htt_content", len, buf);
      dt_setinteger(xfer,  8, "htt_code", 400);
      close(fd);
      return 400;
    }
    /* compare file date to "If-Modified-Since" value
    ** if file is older, respond "Not Modified" */
    if ((gmt.tm_year+1900 <  year)                ||
        (gmt.tm_mon       <  du_http_month(month)) ||
        (gmt.tm_mday      <  mday)                ||
        (gmt.tm_hour      <  hour)                ||
        (gmt.tm_min       <  min)                 ||
        (gmt.tm_sec       <= sec)) {
      dt_setinteger(xfer,  8, "htt_code", 304);
      close(fd);
      return 304;
    }
  }

  /* read file, set content */
  do {
    bytes = read(fd, buf, len);
    if (0 < bytes)
      dt_appendstring(xfer, 11, "htt_content", bytes, buf);
    tbytes += bytes;
  } while ((0 < bytes) || ((-1 == bytes) && (EINTR == errno)));

  /* close file descriptor */
  close(fd);

  /* check for error signal */
  if (-1 == bytes) {
    len = sprintf(buf, "error signal %d (%s)", errno, strerror(errno));
    dt_setstring(xfer, 11, "htt_content", len, buf);
    dt_setinteger(xfer,  8, "htt_code", 500);
    return 500;
  }

  /* get MIME type from path */
  if(NULL != (ptr = strchr(path+1, '/')))
    if(NULL != (ptr = strchr(ptr+1, '/')))
      dt_setstring(xfer, 13, "htt_mime_type", ptr-path-1, path+1);

  /* set status code to success */
  dt_setinteger(xfer,  8, "htt_code", 200);
  return 200;
}
```

```
132. ....
133. ....
134. ....
135. ....
136. ....
137. ....
138. ....
139. ....
```

This code has some features worth mentioning:

◊ Line 32: The `getenv` system call must succeed, otherwise the response is set to `500 Internal Server Error`".
◊ Line 37: The document root directory is set relative to the DY_ROOT environment variable.
◊ Line 41: The URL path must not contain "../", otherwise the response is set to "`404 Not Found`". This is a basic security check on the filename to prevent access outside of the document root directory.
◊ Line 48: The `filename` local variable must be long enough to contain the name of the file, otherwise the response is set to "`500 Internal Server Error`". Changing the local variable memory from static to dynamic is a reasonable alternative, as it supports very long filenames provided that the system can allocate the memory space.
◊ Line 55: The absolute filename is constructed from the DY_ROOT environment variable, document root directory and the URL path. Any arguments after the filename are ignored.
◊ Line 66: The `fstat` system call must succeed, otherwise the response is set to "`500 Internal Server Error`".
◊ Line 75: If the file size empty, the response is set to "`204 No Content`".
◊ Line 82: If the `gmtime_r` and `strfime` system calls succeed, the `htt_modified` property is set, and the "Last-Modified" header is sent in the response.
◊ Line 87: If the `htt_since` property is not NULL (corresponding to the "If-Modified-Since" header), the value of the `htt_since` property is compared to the date of the file.
◊ Line 91: If the date value doesn't specify GMT as the timezone, the response is set to "`400 Client Error`". This is a minimal check on the data for conformance to the RFC 1123 specification.
◊ Line 100: If the file is older than the date specified by the `htt_since` property, the response is set to "`304 Not Modified`".
◊ Line 113: The interrupt signal is ignored while the file is read and set into the `htt_content` property.
◊ Line 124: If a signal other than he interrupt signal ocurrs during reading, the response is set to "`500 Internal Server Error`".
◊ Line 134: The MIME type of the file is implied by the first two subdirectories of the URL path. For example, if the path is `text/html/plain-index.html`, the (implied) MIME type is `text/html`.
◊ Line 137: If execution reaches the end of the routine, the response is set to "`200 OK`".

This source code is part of the `libdu.a` library.

Now, let's construct an example program that uses the `du_http_file` routine to serve files from disk:

Figure 20.5.3-2. Program Source Code: File I/O

```
1. #include <stdio.h>
```

```
#include "dt_smiles.h"
#include "dt_http.h"
#include "du_http.h"

/** my_handler - serve files from disk
*/
void my_handler(dt_Handle xfer) {
  du_http_file(xfer, fd);
}

/** main - calls entry point for dual-purpose server & CGI
*/
int main(int argc, char **argv) {
  return du_http_main(argc,argv);
}
```

This code is available as http-file.c in the "Contrib" area of Daylight Software and made with the command:

```
cd $DY_ROOT/contrib/src/http
make http-file
```

Now, let's execute the the program as a CGI:

```
./http-file << EOF
GET /text/html/index-plain.html HTTP/1.0
EOF
```

You should see output HTML code that contains:

```
<H2>Index of /text/plain Subdirectory</H2>

<UL>
<LI><A HREF=/text/plain/AsImplemented.txt>
  The Original HTTP as defined in 1991 </A> (HTTP 0.9)
<LI><A HREF=/text/plain/draft-coar-cgi-v11-03.txt>
  The WWW Common Gateway Interface Version 1.1 </A> (CGI 1.1)
<LI><A HREF=/text/plain/rfc822.txt>
  RFC 822: Standard for the Format of ARPA Internet Text Messages </A> (date for
mat)
<LI><A HREF=/text/plain/rfc1123.txt>
  RFC 1123: Requirements for Internet Hosts -- Application and Support </A> (dat
e format)
<LI><A HREF=/text/plain/rfc1520.txt>
  RFC 1521: MIME (Multipurpose Internet Mail Extensions) </A>
<LI><A HREF=/text/plain/rfc1945.txt>
  RFC 1945: Hypertext Transfer Protocol -- HTTP/1.0 </A>
<LI><A HREF=/text/plain/rfc2616.txt>
  RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1 </A>
<LI><A HREF=/text/plain/rfc2965.txt>
  RFC 2965: HTTP State Management Mechanism </A> (cookies)
```

The program received the request, got the file (`/text/html/index-plain.html`), set the MIME type as implied from the URL path (`text/html`), and responded with the contents of the file. The file is a prepared index of the `/text/plain` subdirectory, which contains links to public documents pertaining to HTTP specifications used to develop the HTTP Toolkit. If you're not sure if your program output is correct, compare it to http-file.ref (see "Contrib" area of Daylight Software), which contains the complete output from the program.

20.5.4. Scripting with Java

As the name implies, JavaScript is a scripting language for Java. One way to use JavaScript in HTML is to specify a `<SCRIPT>` tag that defines the script and an `<INPUT>` button to invoke the script. Using the toolkit, we can do this by setting the `htm_head_script` property and specifying the *button* argument to the `du_http_form` call:

Figure 20.5.4. Program Source Code: Canonical JavaGrins

```
#include <stdio.h>
#include "dt_smiles.h"
#include "dt_http.h"
#include "du_http.h"

/* my_handler - get POST data and set response
*/
static void my_handler(dt_Handle xfer) {
  dt_Handle string;
  dt_String value = NULL;
  dt_Integer vlen = 0;
  char *name = "SMILES", *type = "TEXT";
  char *encoding = "application/x-www-form-urlencoding";
  char *script   = "LANGUAGE=JavaScript SRC=/application/x-javascript/loadJavaGri
  char *button   = "VALUE=JavaGrins ONCLICK=makeNewWindow('smilesForm.smiles')";

  /* serve files from disk, except for special POST URL ("/") */
  path = dt_string(&plen, xfer, 8, "htt_path");
  if (1 < plen) {
    du_http_file(xfer);
    return;
  }

  /* set return code and mime type */
  dt_setinteger(xfer, 8, "htt_code", 200);
  dt_setstring (xfer, 8, "htt_mime_type", 9, "text/html");

  /* set script and button */
  dt_setinteger(xfer, 12, "htm_autohtml", 1);
  dt_setstring (xref, 15, "htm_head_script", strlen(script), script);

  /* get POST data */
  if (NULL_OB != (string = du_http_post(xfer, strlen(name), name)))
    value = dt_string(&vlen, string, 9, "htt_value");

  /* write form */
  du_http_form(xfer, name, type, encoding, "/", button, vlen, value);

  /* parse smiles and canonicalize */
  if (NULL != value ) {
    dt_Handle molecule;
35.     dt_String cansmiles = "invalid";
36.     dt_Integer clen = 7;
37.     if (NULL_OB != (molecule = dt_smilin(vlen, value)))
38.       cansmiles = dt_cansmiles(&clen, molecule, 1);
39.     /* set response */
40.     dt_appendstring(xfer, 11, "htt_content", 22, "\n<P>Canonical SMILES: ");
41.     dt_appendstring(xfer, 11, "htt_content", clen, cansmiles);
42.     dt_appendstring(xfer, 11, "htt_content",  1, "\n");
        dt_dealloc(molecule);
43.   }
44. }
45.
46. /* main - calls dual-purpose server & CGI
    */
47. int main(int argc, char **argv) {
48.   return du_http_main(argc,argv);
49. }
50.
51.
52.
53.
54.
55.
56.
57.
58.
59.
```

This program adapted from "Canonical SMILES" (Figure 20.5.1.1). The key additions are:

◊ Line 14: The JavaScript language and source are defined.
◊ Line 15: The invocation button is defined.
◊ Line 18: URLs other than "/" are processed as a file from disk.
◊ Line 29: The `htm_autohtml` property is turned on.
◊ Line 30: The `htm_head_script` property is set.
◊ Line 37: The invocation button is a parameter to `du_http_form`.

This code is available as http-javagrins.c in the "Contrib" area of Daylight Software and is made with the command:

```
cd $DY_ROOT/contrib/src/http
make http-javagrins
```

Since JavaGrins is a graphical drawing tool, testing this program requires interaction with a browser. When you click on the button labeled "JavaGRINS", the drawing tool should "pop-up". You will need the `dayutilserver` license to convert drawings to SMILES. For more information on JavaGrins or the `dayutilserver`, see Daylight's *JavaGrins User Guide* at *http://www.daylight.com/dayhtml/doc/java/javagrins.html*.

Nongraphically, we can test that the JavaScript language, script, and invocation button are included in the HTML output. This will test that the `htm_autohtml` and `htm_head_script` properties and that the *button* parameter to the `du_http_form` call works properly. So, let's execute the the program as a CGI:

```
./http-javagrins << EOF
```

```
GET / HTTP/1.0
EOF
```

You should see output HTML code that contains:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 1.1//EN">
<HTML>
<HEAD>
<SCRIPT LANGUAGE=JavaScript SRC=/application/x-javascript/loadJavaGrins.js></SCRIPT>
</HEAD>
<BODY>

<P><FORM METHOD=POST NAME=form ACTION=/ ENCTYPE=application/x-www-form-urlencoding>
SMILES: <INPUT NAME=SMILES TYPE=TEXT VALUE=>
<INPUT TYPE=BUTTON VALUE=JavaGrins ONCLICK=makeNewWindow('smilesForm.smiles')>
<INPUT TYPE=SUBMIT NAME=button>
</FORM>

</BODY>
</HTML>
```

We see that a non-zero `htm_autohtml` property value makes the toolkit output HTML tags before and after the form. The `htm_head_script` property value is in the SCRIPT tag and the *button* parameter to the `du_http_form` call is in the form. If you're not sure if your program output is correct, compare it to http-javagrins.ref (see "Contrib" area of Daylight Software), which contains the complete output from the program.

20.5.5 Formatting With Automatic HTML

Section 20.4.4 describes HTML presentation properties. In this section, we'll demonstrate use of the properties and look at the HTML output that is produced:

Figure 20.5.5. Program Source Code: Automatic HTML

```
#include <stdio.h>
#include "dt_smiles.h"
#include "dt_http.h"
#include "du_http.h"

/* my_handler - demonstration of automatic HTML formatting
*/
void my_handler(dt_Handle xfer) {
  dt_String method, path, protocol;
  dt_Integer mlen, plen, rlen;

  method   = dt_string(&mlen, xfer, 10, "htt_method");
  path     = dt_string(&plen, xfer,  8, "htt_path");
  protocol = dt_string(&rlen, xfer, 12, "htt_protocol");

  dt_setinteger   (xfer,  8, "htt_code",      200);
  dt_setstring    (xfer, 13, "htt_mime_type",   9, "text/html");
  dt_setstring    (xfer, 11, "htt_content",    12, "Hello World!");
  dt_appendstring (xfer, 11, "htt_content",    12, "\n\nMethod:   ");
  dt_appendstring (xfer, 11, "htt_content", mlen, method);
  dt_appendstring (xfer, 11, "htt_content",    11, "\nPath:     ");
18.   dt_appendstring (xfer, 11, "htt_content", plen, path);
19.   dt_appendstring (xfer, 11, "htt_content",   11, "\nProtocol: ");
20.   dt_appendstring (xfer, 11, "htt_content", rlen, protocol);
21.   dt_appendstring (xfer, 11, "htt_content",    1, "\n");
22.   /* auto-HTML formatting */
23.   dt_setinteger   (xfer, 12, "htm_autohtml",      1);
24.   dt_setstring    (xfer, 17, "htm_body_bg_color", 6, "e0e0e0");
25.   dt_setstring    (xfer, 14, "htm_head_title",   21, "Auto-HTML Format Demo");
26.   dt_setstring    (xfer, 10, "htm_prefix",       191,
27.                    "<TABLE BORDER CELLSPACING CELLPADDING=4 WIDTH=100%><TR>\n"
28.                    "<TD ALIGN=CENTER WIDTH=100%>\n"
29.                    "   <A HREF=/>Auto-HTML Format Demo</A>\n"
30.                    "<TD ALIGN=CENTER>\n"
31.                    "   <A HREF=mailto:info@daylight.com>\n"
32.                    "   info@daylight.com</A>\n"
33.                    "<TD ALIGN=CENTER VALIGN=MIDDLE WIDTH=100%>\n"
34.                    "   <A HREF=http://www.daylight.com>\n"
35.                    "   Daylight Chemical Information Systems, Inc.</A>\n"
36.                    "<TD ALIGN=CENTER VALIGN=MIDDLE>\n"
37.                    "   <I>Daylight<BR>");
38.   ver = dt_info(&vlen, NULL_OB, "toolkit_version");
39.   dt_appendstring (xfer, 11, "htm_postfix", vlen, ver);
40.   dt_appendstring (xfer, 11, "htm_postfix", 13, "</I>\n</TABLE>");
41. }
42.
43. /* main - calls entry point for dual-purpose server & CGI
44. */
45. int main(int argc, char **argv) {
46.   return du_http_main(argc,argv);
47. }
48.
49.
50.
51.
52.
53.
54.
55.
56.
57.
```

This program is adapted from "Hello World!" (Figure 20.3.3). The key additions are:

◊ Line 27: Automatic HTML formatting is turned on.
◊ Line 28: The background color is set to light gray.
◊ Line 29: The title is set to "Auto-HTML Format Demo".
◊ Line 30: The page content is prefixed with a table.
◊ Line 34: The page content is postfixed with a table.

This code is available as http-autohtml.c in the "Contrib" area of Daylight Software and is made with the command:

```
cd $DY_ROOT/contrib/src/http
make http-autohtml
```

Now, let's execute the the program as a CGI:

```
./http-autohtml << EOF
GET / HTTP/1.0
EOF
```

You should see output HTML code that contains:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 1.1//EN">
<HTML>
<HEAD>
<TITLE>Auto-HTML Format Demo</TITLE>
</HEAD>
<BODY BACKGROUND=image/gif/sunlogo-gray.gif>

<P><TABLE BORDER CELLSPACING CELLPADDING=4 WIDTH=100%><TR>
<TD ALIGN=CENTER WIDTH=100%>
  <A HREF=/>Auto-HTML Format Demo</A>
<TD ALIGN=CENTER>
  <A HREF=mailto:info@daylight.com>
  info@daylight.com</A>
<TD ALIGN=CENTER VALIGN=MIDDLE WIDTH=100%>
  <A HREF=http://www.daylight.com>
  Daylight Chemical Information Systems, Inc.</A>
<TD ALIGN=CENTER VALIGN=MIDDLE>
  <I>Daylight<BR>4.81</I>
```

If you're not sure if your program output is correct, compare it to http-autohtml.ref (see "Contrib" area of Daylight Software), which contains the complete output from the program.

20.5.6 Adding Color to GIF Images

Here's a neat trick for adding color to GIF images. This is useful for using one image in many HTTP services and distinguishing them by the color of its image. This technique uses red, green, and blue arguments from the URL and works on GIF images that are of type grayscale:

Figure 20.5.6-1. Example Source Code: gif2rgb

```
/* du_http_gif2rgb - add color to a GIF image
*/
void du_http_gif2rgb(char *gif, int red, int green, int blue) {
  int i, byte[7] = { 19, 25, 22, 13, 28, 16, 31 };

  for (i = 0; i < 7; i++) {
    gif[byte[i]]    += red;
    gif[byte[i]+1]  += green;
    gif[byte[i]+2]   = blue;
  }
}
```

```
....
```

This source code is part of the `libdu.a` library.

Now, let's construct an example program that uses the `du_http_gif2rgb` routine to add color to a GIf image:

Figure 20.5.6-2. Program Source Code: Color GIF

```
#include <stdio.h>
#include "dt_smiles.h"
#include "dt_http.h"
#include "du_http.h"

/* my_handler - serve files from disk
*/
void my_handler(dt_Handle xfer) {
  dt_String path, type, args, gif;
  dt_Integer len;
  int red, green, blue;

  du_http_file(xfer);

  /* check for GIF */
  type = dt_string(&len, xfer, 13, "htt_mime_type");
  if ((9 == len) && (0 == memcmp("image/gif", type, 9))) {

    /* check for RGB argument in URL */
    path = dt_string(&len, xfer, 8, "htt_path");
    for (args = path; args <= path+len-4; args++)
      if (0 == memcmp(args, "+rgb=", 5))
        break;

    if (args <= path+len-5) {
      /* get RGB 8-bit values */
      sscanf(args+5, "%2x%2x%2x", &red, &green, &blue);
      if (0 != (red + green + blue)) {
/* get GIF image */
        if (NULL != (gif = dt_string(&len, xfer, 11, "htt_content")))
  /* add color */
          du_http_gif2rgb(gif, red, green, blue);
      }
    }
  }
}

/* main - calls entry point for dual-purpose server & CGI
*/
int main(int argc, char **argv) {
  return du_http_main(argc,argv);
}
```

```
  42. ....
```

This code is available as http-color.c in the "Contrib" area of Daylight Software and made with the command:

```
cd $DY_ROOT/contrib/src/http
make http-color
```

To invoke RGB coloring of a GIF, add "`+rgb=`" and a 6-character hexidecimal string to a GIF URL. The first 2 character are for red, the next two for green, and the last two for blue. For example, the URL `/image/gif/sunlogo-gray.gif+rgb=08010f` adds 8 parts red, 1 part green, and 15 parts blue to make a faint purple image.

20.5.7 Avoiding Run-Time Problems With Compiler Definitions

The consequence of a mispelled literal at run-time often costs more time to solve than at compile-time. This section is dedicated to a simple, yet effective, technique that uses property name definitions to enable the compiler to detect a misspelling instead of experiencing a run-time problem.

Given the numerous named properties in this toolkit, the probability of misspelling a string literal, i.e., "htt_data" instead of "htt_date" is quite high. From the compiler point-of-view, a misspelled literal is syntactically correct, so no problem is detected. At run-time, a problem arises from use of a misspelled property name:

```
        date = dt_string(&length, xfer, 8, "htt_data")
```

This returns NULL (and length equal to -1). Lack of return value checking (as is often the case) can make this problem difficult to solve (I've actually misspelled `htt_date` a couple times). If you're good, you'll find the problem quickly, otherwise, this misspelling problem may cost many minutes or perhaps hours to characterize and correct. In `dt_http.h`, each property name is defined:

```
        #define DX_HTT_DATE           "htt_date"
```

Uuse of the definition (`DX_HTT_DATE`) instead of the literal ("`htt_date`") effectively enables the compiler to detect a misspelling:

```
        date = dt_string(&length, xfer, strlen(DX_HTT_DATE), DX_HTT_DATA);
```

This produces a compiler error such as:

```
"my_http.c", line 11: undefined symbol: DX_HTT_DATA
```

Using property name definitions can save significant time during development of HTTP Toolkit programs. Further, with some compilers (i.e., GCC), the `strlen` call is substituted with the actual length of the string literal at compile-time, so there's no cost in including `strlen` as an argument to `dt_string`. Further, specifying the string length and literal in a `dt_string` call (`strlen(DX_HTT_DATA), DX_HTT_DATA)` creates the possibility that the two interdependent arguments may be inconsistent:

```
        date = dt_string(&length, xfer, strlen(DX_HTT_DATE), DX_HTT_PATH);
```

Defining a macro like:

```
#define DU_PROP(x) strlen(x), x
```

Using it in calls to `dt_string`:

```
date = dt_string(&length, xfer, DU_PROP(DX_HTT_DATE));
```

This averts the problem, or at least makes programming with property names a bit more convenient.

20.5.8 Technical Specifications, Methods, Headers, and Status Codes

This toolkit is compliant with HTTP/1.0 and adheres to specifications in the the following references:

1. Gourley, D.; Totty, B., "*HTTP: The Definitive Guide*", First Edition, O'Reilly & Associates, Inc., September 2002.
2. Coar, K., *"The WWW Common Gateway Interface Version 1.1"*, Internet Draft, June 1999. (*http://cgi-spec.golux.com/draft-coar-cgi-v11-03.txt*)
3. Berners-Lee, T.; Fielding, R.; Frystyk, H., *"Hypertext Transfer Protocol -- HTTP/1.0"*, Request for Comments: 1945, May 1996. (*http://www.ietf.org/rfc/rfc1945.txt*)
4. Gundavaram, S., "*CGI Programming on the World Wide Web*", First Edition, O'Reilly & Associates, Inc., March 1996.
5. Berners-Lee, T., *"The Original HTTP as defined in 1991"*, W3C, 1991. (*http://www.w3.org/Protocols/HTTP/AsImplemented.html*)

The following methods have an integer definition in `dt_http.h`:

- GET
- HEAD
- POST

The following headers have a named property defined in `dt_http.h`:

- General
    - Date
    - Pragma
- Request
    - Authorization
    - From
    - If-Modified-Since
    - Referer
    - User-Agent
- Response
    - Location
    - Server
    - WWW-Authenticate
- Content
    - Allow
    - Content-Encoding
    - Content-Length
    - Content-Type
    - Expires
    - Last-Modified

The following status codes are recognized in `dt_http_put`:

- 200 OK
- 201 Created
- 202 Accepted
- 204 No Content
- 301 Moved Permanently
- 302 Found
- 304 Not Modified
- 400 Bad Request
- 401 Unauthorized
- 403 Forbidden
- 404 Not Found
- 500 Internal Server Error
- 501 Not Implemented
- 502 Bad Gateway
- 503 Service Unavailable

21. Reentrant Toolkit Interface

21.1 Introduction

Beginning with version 4.93, the Daylight toolkits can be used effectively in a multi-threading environment. There are several driving forces for this development:

- Database searching. In database systems such as Daycart and Merlinserver multithreading can be used to seamlessly provide increased search throughput and performance improvement.
- Java. Java supports full multithreading in its object model and a robust toolkit interface to Java should be multithreaded. Previous Java wrappers for the Daylight toolkits were limited by the concurrency issues within the Daylight toolkit.
- Pthreads. The POSIX thread interface (which includes threads, mutexes, conditional variables, signal handling) provides cross-platform capabilities for developing multithreaded applications. This environment allows standard programming methods to be applied for multithreading applications.

With 4.93 we are providing a general, reentrant, multithreading interface to the Daylight toolkits via POSIX threads. The multithreading interface does not substantially change the current external toolkit interface and causes minimal impact on performance of single-threaded toolkit programs.

21.2 Data Issues

Above and beyond the normal programming concerns, the main additional issue which one must be aware of when writing multithreading toolkit programs is the potential need to share objects and other resources within the toolkits across multiple threads.

Because the Daylight Toolkit provides an opaque object model (the internal structure and implementation of the objects is not visible to the programmer) we must provide rules to guide the sharing of objects. There are two main programming models which can be used in a multithreaded program. In both models, however, there are some common features.

First, all error-handling and error queues are implemented on a per-thread basis. That is, if a toolkit function results in an error, that error will be placed on the error queue for the thread which ran the function. The error will not be visible to any other threads. The functions dt_errors(3), dt_errorworst(3), dt_errorsave(3), and dt_errorclear(3) operate on the current threads error queue only.

Heap data (eg. from malloc()) is shared across the entire process. This is as expected for a multithreaded program, however the implications for Daylight Toolkit objects are worth mentioning. All of the internal toolkit object implementations use heap data and the toolkits perform large numbers of malloc() and free() calls. Hence, the performance of the system malloc library is critical to overall throughput of a multithreaded Daylight toolkit program. On some platforms it is desirable to use an alternative malloc library instead of the default system malloc library. On Solaris the library /usr/lib/libmtmalloc.a is optimized for multithreading programs and improves performance significantly. On Linux and SGI the default system malloc library gives good performance.

The other issue with heap data is strings within the toolkit. Most of the toolkit functions which return strings (eg. dt_cansmiles()) are actually returning pointers to strings which are owned by the object itself. It is valid to use these strings across threads, however one must make sure that the object continues to exist within the program.

```
int funca(char *arg)
{
  pthread_t tid;

  dt_Handle mol;
  dt_String str;
  dt_Integer slen;

  mol = dt_smilin(strlen(arg), arg);
  str = dt_cansmiles(&slen, mol, 1);

  pthread_create(&tid, NULL, funcb, str);

  dt_dealloc(mol);
  return 0;
}
```

In the above example, a new thread calling 'funcb()' is created. The problem here is that the canonical SMILES string (str), which is passed into the child thread, gets removed when the molecule object is deallocated. It is likely that funcb() will fail as soon as it tries to use the string. In this case it would be better to duplicate 'str' and pass the duplicate to the child thread.

21.3 Per-Thread Object Model

The first model for programming uses per-thread objects. Each thread maintains its own handle table for dispatching handles to their underlying objects. The objects are not shared across threads. This is the simpler model to implement as no locking of objects needs to be performed.

The basic program requirements are as follows:

- The function dt_mp_initialize(DX_MP_PER_THREAD_HANDLES) must be called before any toolkit objects are allocated. It is generally best to call this function from the main thread during startup.

- Each child thread created in the program will only have access to objects created within that thread. It is important to note that every thread *may* use a given handle ID to represent a different local object. That is, every thread has it's own internal table of handle IDs and the same ID numbers will be used by multiple threads. So if a programmer takes a handle allocated in one thread and attempts to access it in second thread, the second thread will find that the handle is either invalid or refers to a different object (local) object.

  If an object is required in multiple threads it is necessary to create that object from it's string representation in each thread. Typically one thread can create a string from the object and pass that string to the other threads that need the object. They will then instantiate a local object.

A simple example is the smarts_filter_mt.c program, which reads SMILES on stdin and writes any SMILES which match a given SMARTS query to stdout.

```
void *do_smarts_forever(void *arg)
{
  static const int ok = 1;
  static const int fail = -1;
  char      line[MAXSMI];
  dt_Handle mol, pattern, pathset;
  char *smarts = (char *)arg;

  pattern = dt_smartin(strlen(smarts), smarts);
  if (pattern == NULL_OB)
    {
      fprintf(stderr, "Can't parse SMARTS  in child thread\n", smarts);
      return((void *)&fail);
    }

  while (!feof(stdin))
    {
      if (!gets(line))
        return((void *)&ok);

      mol = dt_smilin(strlen(line), line);
      if (mol != NULL_OB)
        {
          pathset = dt_match(pattern, mol, TRUE);
          if (pathset != NULL_OB)
            {
              dt_dealloc(pathset);
              printf("%s\n", line);
            }
         }
      dt_dealloc(mol);
    }
  return((void *)&ok);
}

#define THR_COUNT 4

int main(int argc, char *argv[])
{
  pthread_t tid;
  int i;

  dt_mp_initialize(DX_MP_PER_THREAD_HANDLES);
  /*** Get SMARTS from command line ***/

  if (2 != argc)
    {
      fprintf(stderr, "usage: %s SMARTS\n", argv[0]);
      exit(1);
    }

  for (i = 0; i <THR_COUNT; i++)
    pthread_create(&tid, NULL, do_smarts_forever, (void *)&argv[1]);
  return (0);
}
```

The main points illustrated in the smarts_filter example are:

- Each thread creates it's own local 'pattern' object from the given SMARTS string rather than attempting to share a single pattern.
- The coordination of input and output streams (gets() and printf()) is handled by the stdio library. Since the granularity of the I/O is line-at-a-time the stdio library makes sure that the I/O is parsed properly. The programmer does not need to do anything special.

21.4 Global Object Model

The second model for programming used global objects. That is, every object allocated within the application is visible to all threads. This model is more complicated to implement as it is necessary for the programmer to synchronize access to any shared objects that are used between threads.

The basic program requirements are as follows:

- The function dt_mp_initialize(DX_MP_GLOBAL_HANDLES) must be called before any toolkit objects are allocated. It is generally best to call this function from the main thread during startup.
- There is a single handle ID namespace, so handles can be passed between threads and reference the same object in both threads. Objects which are created and used locally within a thread do not need special handling; they can be used without any locking or synchronization.
- If an object is required in multiple threads the program must take care to only allow one thread to be accessing the object and it's methods at a time. The functions dt_mp_lock(), dt_mp_trylock(), and dt_mp_unlock() are provided as a convenience for these operations.

The analogous smarts_filter_mt.c is shown below, where the SMARTS pattern is shared between all threads.

```
static dt_Handle pattern;

void *do_smarts_forever(void *arg)
{
  static const int ok = 1;
  static const int fail = -1;
  char      line[MAXSMI];
  dt_Handle mol, pathset;

  while (!feof(stdin))
    {
      if (!gets(line))
        return((void *)&ok);
      mol = dt_smilin(strlen(line), line);

      if (mol != NULL_OB)
        {
          dt_mp_lock(pattern);
          pathset = dt_match(pattern, mol, TRUE);
          dt_mp_unlock(pattern);
          if (pathset != NULL_OB)
            {
              dt_dealloc(pathset);
              printf("%s\n", line);
            }
        }
      dt_dealloc(mol);
    }
  return((void *)&ok);
}

#define THR_COUNT 4

int main(int argc, char *argv[])
{
  pthread_t tid;
  int i;

  dt_mp_initialize(DX_MP_PER_THREAD_HANDLES);

  /*** Get SMARTS from command line ***/

  if (2 != argc)
    {
      fprintf(stderr, "usage: %s SMARTS\n", argv[0]);
      exit(1);
    }

  pattern = dt_smartin(strlen(smarts), smarts);
  if (pattern == NULL_OB)
    {
      fprintf(stderr, "Can't parse SMARTS  in child thread\n", smarts);
      exit(1);
    }

  for (i = 0; i <THR_COUNT; i++)
    pthread_create(&tid, NULL, do_smarts_forever, NULL);
  return (0);
}
```

In the above example, the pattern object is created in the parent (main) thread. Each child uses the same pattern object for the dt_match() operation. The pattern object must be locked before executing the dt_match() function and unlocked after the match is complete.

It is important to note that the dt_mp_lock()/dt_mp_unlock() mechanism is a cooperative locking scheme. Every thread which needs to access a shared object must lock it. When one thread locks an object it does not automatically prevent other threads from accessing the object or calling it's methods; the programmer has the responsibility of guarding against access by other threads.

21.5 Object Granularity

Since the toolkit is implemented using an opaque interface, internal behaviors of toolkit functions are not rigorously defined. Only external behaviors are well defined. This is a very pleasant model for Daylight; we are free to use lazy evaluation, to organize internal data however we choose, and to change these internal data organizations at will.

The unfortunate side effect of this opaque interface is that we can't precisely describe the results of a given modification of one object. A seemingly simple modification to an object may have far-reaching impacts upon other data structures within the toolkit. Given that the key to a multithreaded library implementation is the control of data modifications, this leads to a problem. The solution to this problem is simply to define the allowed concurrancy at a larger granularity than otherwise possible.

The basic granularity of access allowed for objects within the toolkit is the object family. The 'object family' is a new concept within the toolkit which refers to a collection of objects which are related to one-another as parent/children or base/derivitives. All objects within an object family will share the same object as their ancestor (dt_ancestor(3)). The ancestor object is the ultimate parent of all of the objects within the family.

If a thread is accessing/manipulating any object within an object family it is not safe for another thread to be accessing/manipulating any other objects within that object family. Because we can guarantee that side-effects caused by one thread manipulating an object will be contained within the object family, this works out to allow well-defined behavior for multithreaded programs.

There is one complication however, which is toolkit functions which depend on multiple objects (eg. dt_match()). In those cases each thread must have exclusive access to all of the object families needed for the function in order to be thread safe. In the case of dt_match() the thread must either lock both the pattern object and the target object. Alternatively, if either object is known to be local to a thread (eg. in the smarts_filter_mt.c example above the molecule is local to the thread) then it is not necessary to lock that object.

21.6 Thread Safety versus Reentrancy

The toolkit is reentrant, not thread-safe. It is the responsibility of the programmer to take the tools and write multithreading applications. If desired, one can implement a heavy-handed thread-safe toolkit interface as follows. Every toolkit function could be wrapped with a layer which locks the object family before operating on an object, then this wrapper layer would provide a completely thread-safe toolkit API. This wrapper would probably have fairly poor performance.

```
dp_xxx(ob)
{

  dt_mp_lock(ob);
  rc = dt_xxx(ob);

  if (type(rc) is string)
    duplicate string;
  dt_mp_unlock(ob))
  return rc or duplicate string;
}
```

Note that in the above wrapper any returned strings are duplicated. This eliminates the previously mentioned warning about accessing strings within shared objects.

Nothing prevents a programmer from misusing threads, or from accessing the same object from multiple threads. Only if the programmer always gets a lock for the object family before modification can he be guaranteed to have exclusive access to the object. If one or more threads does not obey this convention, then the program may not be thread-safe.

21.7 Limitations

There are several areas of the toolkit which are not reentrant and can not be used by multiple threads at the same time.

The database access toolkits (Thor and Merlin client toolkits) are not reentrant and can not be used by more than one thread in a program at a time. It is possible for a multithreaded program to include these toolkits provided that access to both toolkits is serialized completely within the application.

The Rubicon toolkit is not reentrant and can not be used by more than one thread at a time. It is possible for a multithreaded program to include Rubicon functionality provided that access to the toolkit is serialized completely within the application.

Within the depict toolkit, use of the dt_depict() function and drawing library must be serialized across the entire application. It is not possible for multiple dt_depict() calls to be in progress in different threads at the same time since both will result in the invocation of global drawing library functions. In practice, the drawing library that a user implements must reference one or more global variables (locked along with the dt_depict() function) which can provide the local thread context necessary for the desired drawing operations.

Several obsolete toolkit functions are not reentrant. These include dt_smilinerrtext(), dt_alloc_fp(), dt_fp_fingerprint(), dt_fp_fold(), dt_fp_mindensity(), dt_fp_minsize(), dt_fp_size(), dt_fp_setmindensity(), dt_fp_setminsize() and dt_fp_setsize(). These functions had previously been made obsolete because their API did not fit well within the Daylight toolkit model and should not be used for new programs.