

Transbase[®]
Routines for NUMERIC and DATETIME

Transaction Software GmbH
Willy-Brandt-Allee 2
D-81829 München
Germany
Phone: +49-89-62709-0
Fax: +49-89-62709-11
Email: info@transaction.de
<http://www.transaction.de>

Version 6.8.1.40
November 02, 2010

Contents

1	Routines for Numerics	3
1.1	getprec	3
1.2	getscale	4
1.3	get sign	4
1.4	fixmkzero	4
1.5	fixiszero	5
1.6	fixminus	5
1.7	fixcopy	5
1.8	fixcmp	5
1.9	double_fix	6
1.10	int4_fix	6
1.11	scan_fix	6
1.12	fix_fix	7
1.13	fix_double	7
1.14	fix_int_4	8
1.15	fix_string	8
1.16	fixadd	8
1.17	fixsub	9
1.18	fixmul	9
1.19	fixdiv	9

2	Routines for Datetime and Timespan	11
2.1	Conversion into String Representation	12
2.1.1	Datetime to String Functions	12
2.1.2	Datetime Format Function	14
2.1.3	Timespan to String Function	15
2.1.4	Timespan Format Function	16
2.2	Computations with Datetime and Timespan	17
2.2.1	set_current	18
2.2.2	dt_current	18
2.2.3	dt_weekday	18
2.2.4	dt_copy	19
2.2.5	ts_copy	19
2.2.6	dt_cast	19
2.2.7	ts_cast	19
2.2.8	dt_cmp	20
2.2.9	ts_cmp	20
2.2.10	ts_add	20
2.2.11	ts_sub	20
2.2.12	ts_mul	21
2.2.13	ts_div	21
2.2.14	dt_sub	21
2.2.15	dt_ts_add	21
2.2.16	dt_ts_sub	22
2.2.17	ts_change_sign	22
2.2.18	dt_str	22
2.2.19	ts_str	22
2.2.20	dt_check	23
2.2.21	ts_check	23
2.3	Constructing Values of Type Datetime and Timespan	23
2.3.1	The C-Type Definitions of Datetime and Timespan	23
2.3.2	dtsethighf, dtsetlowf	24
2.3.3	Sample Program	25

Chapter 1

Routines for Numerics

This section describes routines for processing host variables of type Numeric in the ESQL program. The data type Numeric is defined as a structure. All the routines described below are defined in the library `tbx.a` which must be linked to the precompiled and compiled program.

All these functions have pointers to the data type Numeric as parameters. For shortness, the textual explanation of the routines abstracts from this fact and refers to the parameters as if they were passed by value. Furthermore, throughout the following examples the notation `<xy.z>` stands for a pointer to a Numeric variable which holds the value `xy.z`.

1.1 `getprec`

Syntax:

```
int getprec (num)
Numeric *num;
```

Effect: Returns precision of `num`, i.e. the number of digits of `num` without leading zeros and trailing zeros in front or after the fractional point, resp.

Example:

`getprec (<521.1>)` returns 4.

1.2 getscale

Syntax:

```
int getscale (num)
Numeric *num;
```

Effect: Returns scale of num, i.e. the number of digits after the fractional point without trailing blanks.

Example:

```
getscale (<11.102>) returns 3
getscale (<.102>) returns 3
getscale (<0.3450>) returns 4
getscale (<12t3.>) returns 0
```

1.3 get sign

Syntax:

```
int getsign (num)
Numeric *num
```

Effect: Returns 0 if num is zero or positive, else 1.

Example:

```
getsign (12.123) returns 0
getsign (.0) returns 0
getsign (-1.1) returns 1
```

1.4 fixmkzero

Syntax:

```
Numeric *fixmkzero (num)
Numeric *num
```

Effect: Stores value 0 into num. Returns num.

This function is equivalent to `int4_fix (0L, 0, num)`.

1.5 `fixiszero`

Syntax:

```
int fixiszero (num)
Numeric *num;
```

Effect: Returns TRUE (1) if num has value 0 else FALSE (0).

1.6 `fixminus`

Syntax:

```
Numeric *fixminus (source, target)
Numeric *source, *target;
```

Effect: Value of source multiplied by -1 is stored into target.

Returns target.

1.7 `fixcopy`

Syntax:

```
Numeric *fixcopy (source, target)
Numeric *source, *target
```

Effect: Copies source to target. Returns target.

1.8 `fixcmp`

Syntax:

```
int fixcmp (n1, n2)
Numeric *n1, *n2;
```

Effect: Compares n1 and n2. Returns negative value if $n1 < n2$, 0 if $n1 = n2$, positive value if $n1 > n2$.

1.9 double_fix

Syntax:

```
Error double_fix (d, outscale, num)
double d;
unsigned outscale;
Numeric *num;
```

Effect: Transforms d into a numeric and stores the result into num. Parameter outscale is used as the scale of the result. Returns 0 if no error is encountered. Returns errorcode $\neq 0$ if d is too large to fit in a numeric.

Errors: Overflow

1.10 int4_fix

Syntax:

```
Numeric *int4_fix (ivalue, outscale, target)
Int4 ivalue; unsigned outscale; Numeric *target;
```

Effect: Transforms ivalue into a Numeric and stores it into target. Outscale zeroes are produced behind the fractional point. Returns target.

1.11 scan_fix

Syntax:

```
int scan_fix (s, target, scanned)
char *s; Numeric *target;
unsigned *scanned
```

Effect: Scans a textual representation *s* of a Numeric and stores it into *target*. Input *s* is scanned until the first character is encountered which is '\0' or is not part of a Numeric. The accepted input format is

[blank | tab | newline]* [+ | -] d* [. d*] where at least one digit *d* appears. Returns in parameter "scanned" the number of scanned characters. Function returns 0 if no error (overflow) is encountered.

Errors: Overflow

1.12 fix_fix

Syntax:

```
Numeric *fix_fix (source, outscale, target)
Numeric *source, *destin; unsigned outscale;
```

Effect: If *outscale* is less than the scale of the source, source is rounded on the *outscale*-th digit. If *outscale* is bigger than the scale of source, source is extended by trailing zeros to the indicated *outscale*.

Example:

```
fix_fix (13.345, 2 , target) returns 13.35
fix_fix (13.345, 1 , target) returns 13.3
fix_fix (13.345, 0 , target) returns 13.
fix_fix (13.345, 5 , target) returns 13.34500
```

1.13 fix_double

Syntax:

```
double fix_double (num)
Numeric *num;
```

Effect: Converts *num* to a double and returns it.

1.14 `fix_int_4`

Syntax:

```
int fix_int_4 (num, target)
Numeric *num;
int_4 *target;
```

Effect: Rounds `num` to a `int4` integer and stores it into `target`. Returns error-code $\neq 0$ if significant digits would be lost, else 0.

1.15 `fix_string`

Syntax:

```
char *fix_string (num, target)
Numeric *num; char *target;
```

Effect: Writes a textual representation of `num` into `target`; `target` must point to an appropriately sized buffer. Maximal requested size is `MAXFIX` (a predefined constant). The target representation begins with a space or character `'-'` depending on the sign of `num`, followed by the non-fractional part

(omitted if $0 < |\text{num}| < 1$, `'0'` if `num=0`), followed by `'.'`, followed by the fractional part (omitted if `num` has no fractional part), followed by `'\0'`. Returns `target`.

1.16 `fixadd`

Syntax:

```
int fixadd (num1, num2, target)
Numeric *num1, *num2, *target;
```

Effect: Adds `num1` and `num2` and writes result into `target`. Returns 0 if no error is encountered. Returns $\neq 0$ if overflow occurs.

Errors: Overflow

1.17 fixsub

Syntax:

```
int fixsub (num1, num2, target)
Numeric *num1, *num2, *target;
```

Effect: Subtracts num2 from num1 and writes result into target. Returns 0 if no error is encountered. Returns $\neq 0$ if overflow occurs.

Errors: Overflow

1.18 fixmul

Syntax:

```
int fixmul (num1, num2, target)
Numeric *num1, *num2, *target;
```

Effect: Multiplies num1 and num2 and writes result into target. Returns 0 if no error is encountered. Returns $\neq 0$ if overflow occurs.

Errors: Overflow

1.19 fixdiv

Syntax:

```
int fixdiv (num1, num2, target)
Numeric *num1, *num2, *target;
int __fixdiv (num1, num2, target, outscale)
Numeric *num1, *num2, *target;
int outscale;
```

Effect: Divides num1 by num2 and writes result into target. This routine effectively computes on the basis of doubles. fixdiv creates the result by using the routine double_fix with outscale=10. __fixdiv creates the result by using the routine double_fix with the specified outscale. Returns 0 if no error is encountered.

Errors: Division by 0

Overflow

Chapter 2

Routines for Datetime and Timespan

Transbase SQL supports the rather elaborated types Datetime and Timespan (DT types, for short). As there are no directly corresponding basic C-types, one needs auxiliary methods for the information flow of DT values between Transbase and the application. Transbase offers 2 solutions to this problem.

The first solution consists in explicit transformation of DT values on SQL level. For the retrieval of DT values one can use the selector functions `YY`, `MO`, `DD`, etc. which can break a DT value in its integer valued components (see the description in the TB/SQL manual). Vice versa, to describe application generated DT values in SQL queries (input values in `INSERT` or `UPDATE` queries or comparison values in `SELECT` queries) one can use the SQL literal representation or (for parameters in stored queries) one can supply DT components in integer valued host variables and use the TB/SQL operator `CONSTRUCT` to transform them into a DT value. Consequently applied, the TB/SQL selector and `CONSTRUCT` operators enable to process DT values in the application in a rather simple way.

A more elaborate and efficient solution is offered by using C structures Datetime and Timespan (defined in `tbx.h`) which serve to store and build up DT values in the Transbase internal format. A rich set of functions is supported in the library `tbx.a` to process DT values. The following section of this manual describes these functions. According to the functionality, the description is divided into two parts: The part "Conversion into String Representation" describes a set of functions to transform DT values into various string formats, e.g. for terminal output. The part "Computations with Datetime and Timespan" describes a complete set of functions to perform all computations with DT values as known from TB/SQL. This part assumes that the reader is familiar with the DT descriptions in the TB/SQL manual.

2.1 Conversion into String Representation

This chapter describes a number of conversion routines provided for C application programmers who want to convert `DATETIME` and `TIMESPAN` values into strings for subsequent output. Note that the generated formats are all different from the TB/SQL literal format because the latter is rather syntactic oriented (e.g. `DATETIME[HH:SS] (12:34:50)`) and not suitable for output purposes.

All functions are included within the Transbase runtime library "tbx.a" which anyway has to be linked in each application program for database access. All function names to handle `DATETIME` values are prefixed with ''tb_dt_'' and all `TIMESPAN` handling functions are prefixed with ''tb_ts_'' . All functions operate on pointers to Datetime or Timespan values. For example, if a variable `dt` of type `Datetime[YY:DD]` has been declared in the `DECLARE SECTION` and a database value has been assigned to `dt` via a `FETCH` statement, then `&dt` would be a valid argument to these functions. Note however, pointers passed to the conversion routines must not be `NULL` pointers.

2.1.1 Datetime to String Functions

Syntax:

```
char * tb_dt_datetime (dest, nlp, ptr)
char * dest;
int nlp;
Datetime * ptr;
```

```
char * tb_dt_date (dest, nlp, ptr)
char * dest;
int nlp;
Datetime * ptr;
```

```
char * tb_dt_time (dest, nlp, ptr)
char * dest;
int nlp;
Datetime * ptr;
```

Effect: These functions convert a `Datetime` value (pointed to by "ptr") into the string given by its address "dest". The destination buffer must be large enough to hold the conversion. The conversion is influenced by the "National Language Parameter" `nlp`. Values for `nlp` are: `ISO`, `USA`, and `EUR`. Depending on `nlp`, the function `tb_dt_datetime` converts the datetime value into the following string formats:

ISO 1989-06-15 14:20:12.123

USA: 06/15/1989 2:20:12.123 PM

EUR: 15.06.1989 14:20:12.123

The function `tb_dt_date` converts the datetime value into one of the following string formats:

ISO 1989-06-15

USA: 06/15/1989

EUR: 15.06.1989

The function `tb_dt_time` converts the datetime value into one of the following string formats:

ISO: 14:20:12.123

USA: 2:20:12.123 PM

EUR: 14:20:12.123

0 If the datetime value does not contain all components shown above, only those within the datetime range are given. Note that this may produce also strings as e.g. "23 9:00" if the range of the Datetime value was [DD:MI].

Returns: dest

Example:

```
EXEC SQL BEGIN DECLARE SECTION;  
Datetime[YY:DD] dt;
```

```
EXEC SQL END DECLARE SECTION;
char dest[40];

...
EXEC SQL FETCH C3 INTO ..., :dt, ...;
...
tb_dt_datetime (dest, USA, &dt);
printf (''%s\n'', dest);
...
}
```

2.1.2 Datetime Format Function

Syntax:

```
char * tb_dt_format (dest, format, ptr)
char * dest;
char * format;
Datetime * ptr;
```

Effect: This function converts a Datetime value (pointed to by "ptr") into the string given by its address "dest" according to a user-defined format string (given by "format"). The destination buffer must be large enough to hold the string produced by `tbdtformat`. The format string is built similar to the format string used by the UNIX "date" command. Character sequences `%c` are interpreted by `tb_dt_format` as follows:

`%n` a newline character
`%t` a tab character
`%y` the year (four digits, zero-filled)
`%m` the month (two digits, zero-filled)
`%d` the day (two digits, zero-filled)
`%H` the hour (two digits, zero-filled)
`%M` the minute (two digits, zero-filled)
`%S` the second (two digits, zero-filled)
`%F` the millisecond (three digits, zero-filled) of the given date
`%%` the percent character itself

If a component of the datetime value is referenced which is not defined nothing is interpolated into dest.

Returns: dest

Example:

```
EXEC SQL BEGIN DECLARE SECTION;
Datetime[YY:DD] dt;
EXEC SQL END DECLARE SECTION;
char dest[40];

...
EXEC SQL FETCH C3 INTO ..., :dt, ..;
...
tb_dt_format (dest, 'Date %dth of month %m, %y', &dt);
```

would print e.g. the line:

```
Date 15th of month 06, 1989
```

2.1.3 Timespan to String Function

Syntax:

```
char * tb_ts_timespan (dest, nlp, ptr)
char * dest;
int nlp;
Timespan * ptr;
```

Effect: This function converts a Timespan value (pointed to by "ptr") into the string given by its address "dest". The destination buffer must be large enough to hold the resulting string. Currently, the conversion is not influenced by the "National Language Parameter" nlp. Future versions may support the nlp parameter. The function `tb_ts_timespan` converts the timespan value into strings of the following shape:

1000y 10mo	1000 years and 10 months
1000y	1000 years
10mo	10 months
12d 12:33:12.123h	12 days, 12 hours, 33 minutes, ...
12:33:12.123h	12 hours, 33 minutes, ...

33:12.123mi	33 minutes, 12.123 seconds
12.123s	12.123 seconds
123ms	123 milliseconds

Note that no timespan values are allowed that cross the boundary between months and days.

Returns: dest

Example:

```
EXEC SQL BEGIN DECLARE SECTION;
Timespan ts;
EXEC SQL END DECLARE SECTION;
char dest[40];
...
EXEC SQL FETCH C3 INTO ..., :ts, ..;
tb_ts_timespan (dest, USA, &ts);
}
```

2.1.4 Timespan Format Function

Syntax:

```
char * tb_ts_format (dest, format, ptr)
char * dest;
char * format;
Timespan * ptr;
```

Effect: This function converts a Timespan value (pointed to by "ptr") into the string given by its address "dest" according to a user-defined format string (given by "format"). The destination buffer must be large enough to hold the conversion. The format string is built similar to the format string used by the UNIX "date" command. Character sequences follows:

%n a newline character

%t a tab character

%- the minus sign if timespan is negative, otherwise a blank

%y the number of years of the given timespan
 %m the number of months of the given timespan
 %d the number of days of the given timespan
 %H the number of hours of the given timespan
 %M the number of minutes of the given timespan
 %S the number of seconds of the given timespan
 %F the number of milliseconds of the given timespan
 %% the percent character itself

If a component of the datetime value is referenced which is not defined nothing is interpolated into dest.

Returns: dest

Example:

```

EXEC SQL BEGIN DECLARE SECTION;
Timespan ts;
EXEC SQL END DECLARE SECTION;
char dest[40];

...
EXEC SQL FETCH C3 INTO ..., :ts, ..;
...
tb_ts_format (dest,
              'Time Difference is %y years and %d months'', &ts);
  
```

would print e.g. the line:

```
Time Difference is 2 years and 201 months''
```

2.2 Computations with Datetime and Timespan

For the following functions, the reader is assumed to be familiar with the semantics of Datetime and Timespan and their defined operations from the reading of the corresponding chapters of the Transbase SQL manual.

Many of the datetime and timespan functions return an error code. An error code 0 means no error. If an error code $\neq 0$ is delivered then a textual description of the error is available. Note that the error mechanism is not consistent with the error handling of `EXEC SQL` statements in two respects: first, the pointer variable for the error text is `tb_errtxt` in contrast to `sqlca.tb_errtxt`. Secondly, errors of the datetime and timespan functions are not caught by the `WHENEVER SQLERROR` clause but must be explicitly handled after each call.

2.2.1 `set_current`

Syntax:

```
Error set_current()
```

Effect: Sets and actualizes the current date. Between two calls of this routine, the date delivered by routine `dt_current()` remains the same.

2.2.2 `dt_current`

Syntax:

```
Error dt_current(dt)
Datetime *dt;
```

Effect: Delivers the current date in `dt`. The range is `[YY:MS]`. The current date is that of the time of the last `set_current` call.

2.2.3 `dt_weekday`

Syntax:

```
Error dt_weekday(dt, day)
Datetime *dt;
int *day;
```

Effect: Delivers in `day` the weekday of `dt`. The weekday is encoded as a number from 0 to 6 where 0 means sunday, 1 means monday etc. until 6 which means saturday.

2.2.4 dt_copy

Syntax:

```
void dt_copy(target,source)
Datetime *target, *source;
```

Effect: Copies source to target; Only relevant bytes are copied.

2.2.5 ts_copy

Syntax:

```
void ts_copy(target,source)
Timespan *target, *source;
```

Effect: Copies source to target; Only relevant bytes are copied.

2.2.6 dt_cast

Syntax:

```
Error dt_cast(dtin,dtout,lowf,highf)
Datetime *dtin, *dtout; int lowf, highf;
```

Effect: Casts dtin according to lowf and highf and writes result to dtout. Memory areas of dtin and dtout may overlap or be identical.

2.2.7 ts_cast

Syntax:

```
Error ts_cast(tsin,tsout,lowf,highf)
Timespan *tsin, *tsout; int lowf, highf;
```

Effect: Casts tsin according to lowf and highf and writes result to tsout. Memory areas of tsin and tsout may overlap or be identical.

2.2.8 dt_cmp

Syntax:

```
Error dt_cmp(dt1,dt2)
Datetime *dt1, *dt2;
```

Effect: Compares dt1 and dt2. Returns 0 if they are equal, returns negative (positive) value if dt1 is less (greater) than dt2.

2.2.9 ts_cmp

Syntax:

```
Error ts_cmp(ts1,ts2,result)
Timespan *ts1, *ts2;
int *result;
```

Effect: Compares ts1 and ts2. Returns in result 0 if they are equal, returns in result negative (positive) value if ts1 is less (greater) than ts2. Returncode is <> 0 if timespan values are invalid.

2.2.10 ts_add

Syntax:

```
Error ts_add(ts1,ts2,tsres)
Timespan *ts1, *ts2, *tsres;
```

Effect: Computes sum of ts1 and ts2 and stores it into tsres. Memory area of tsres may overlap with ts1 or ts2.

2.2.11 ts_sub

Syntax:

```
Error ts_sub(ts1,ts2,tsres)
Timespan *ts1, *ts2, *tsres;
```

Effect: Computes difference of ts1 and ts2 and stores it into tsres. Memory area of tsres may overlap with ts1 or ts2.

2.2.12 ts_mul

Syntax:

```
Error ts_mul(ts1,scalar,tsres)
Timespan *ts1, *tsres;
int_4 scalar;
```

Effect: Multiplies ts1 with scalar and stores result into tsres. Memory area of tsres may overlap with ts1.

2.2.13 ts_div

Syntax:

```
Error ts_div(ts1,scalar,tsres)
Timespan *ts1, *tsres;
Int4_ scalar;
```

Effect: Divides ts1 by scalar and stores result into tsres. Memory area of tsres may overlap with ts1.

2.2.14 dt_sub

Syntax:

```
Error dt_sub(dt1,dt2,tsres)
Datetime *dt1, *dt2;
Timespan *tsres;
```

Effect: Computes the difference between dt1 and dt2 and stores the result (Timespan) into tsres.

2.2.15 dt_ts_add

Syntax:

```
Error dt_ts_add(dt,ts,dtres)
Datetime *dt, *dtres;
Timespan *ts;
```

Effect: Adds `ts` to `dt` and stores the result (Datetime) into `dtres`.

2.2.16 `dt_ts_sub`

Syntax:

```
Error dt_ts_sub(dt,ts,dtres)
Datetime *dt, *dtres;
Timespan *ts;
```

Effect: Subtracts `ts` from `dt` and stores the result (Datetime) into `dtres`.

2.2.17 `ts_change_sign`

Syntax:

```
void ts_change_sign(ts)
Timespan *ts;
```

Effect: Changes sign of `ts`

2.2.18 `dt_str`

Syntax:

```
char *dt_str(str,dt)
char *str;
Datetime *dt;
```

Effect: Writes a string representation of `dt` namely the TB/SQL literal representation into `str`. Returns target address.

2.2.19 `ts_str`

Syntax:

```
char *ts_str(str,ts)
char *str;
Timespan *ts;
```

Effect: Writes a string representation of `ts` namely the TB/SQL literal representation into `str`. Returns target address.

2.2.20 `dt_check`

Syntax:

```
Error dt_check(dt)
Datetime *dt;
```

Effect: Checks whether `dt` is a semantically valid datetime and returns an error if not.

2.2.21 `ts_check`

Syntax:

```
Error ts_check(ts)
Timespan *ts;
```

Effect: Checks whether `ts` is a semantically valid Timespan and returns an error if not.

2.3 Constructing Values of Type Datetime and Timespan

In most cases, the functions described so far are called with DT values which are result fields from `SELECT` queries. However, to be self contained, the following chapter describes how to construct values in the C application from integer valued components (analogous to the `CONSTRUCT` operator on SQL level). Note that this procedure is also necessary for evaluation of parameterized queries if one parameter is of type `DATETIME` or `TIMESPAN`.

2.3.1 The C-Type Definitions of Datetime and Timespan

The following type definitions are in the file `tbx.h` which is automatically included in ESQL programs:

2.3. CONSTRUCTING VALUES OF TYPE DATETIME AND TIMESPAN 25

```
typedef struct {
  Int4 qual;
  short val [MAXQUAL];          /* MAXQUAL is 7 */
} Datetime;

typedef struct {
  Int4 qual;
  Int4 val [MAXQUAL];
} Timespan;
```

The component `qual` contains the sign bit and the range description of the value (e.g. `[YY:DD]`). The array `val` contains the values of the single components of the value, e.g. the value `DATETIME[YY:DD][1992-05-21)` contains 3 entries in the `val` array. Note that for saving storage inside Transbase tables, only the relevant components of `Datetime` and `Timespan` values are stored (3 values instead of 7 in our example, i.e. only `val[0]` to `val[2]` are set). Furthermore, the values are stored in ascending weight order (i.e. in the example `val[0]` contains the lowest weighted component 21). See the sample program for a more general example.

2.3.2 dtsethighf, dtsetlowf

Syntax:

```
void dtsethighf(dt,highf)
  Datetime *dt;
  int highf;          /* MS, SS, MI, HH, DD, MO, YY */

void dtsethighf(ts,highf)
  Timespan *ts;
  int highf;          /* MS, SS, MI, HH, DD, MO, YY */

void dtsetlowf(dt,lowf)
  Datetime *dt;
  int lowf;           /* MS, SS, MI, HH, DD, MO, YY */

void dtsetlowf(ts,lowf)
  Timespan *ts;
  int lowf;           /* MS, SS, MI, HH, DD, MO, YY */

void dtsetsign(dt,period)
  Datetime *dt;
  int period;        /* JULS, GREGS */
```

```
void dtsetsign(ts,sign)
Timespan *ts;
int sign;      /* MINUSS, PLUSS */
```

Effect: dtsethighf and dtsetlowf are macros that accept Datetime as well as Timespan values as parameter. They set the "highfield" (highf) and the "lowfield" (lowf) of the value to be constructed. The highf and lowf specify the upperbound and the lowerbound of the range of the value, e.g. in a value of type DATETIME [MO:SS], the highf is MO and the lowf is SS.

dtsetsign is a macro that accepts Datetime as well as Timespan values as parameter. For a timespan value, it sets the sign bit which indicates whether it is positive or negative. For a Datetime value it sets the period, i.e. a flag whether the date is of Julian or Gregorian period. This period flag for datetimes semantically is redundant but must be set such that the interface routines can properly and efficiently work. For ease of use, the interface routine dt_check does not read but sets the period flag, so it can be used as an aid to construct a datetime value (see the sample program).

2.3.3 Sample Program

The following program builds a datetime value and a timespan value from some constants and prints them in the Transbase SQL literal representation.

```
#include <stdio.h>
#include ''tbx.h''
#include ''tborder.h''
short dtf[7] = { 7, 8, 9, 10, 17, 5, 1992 };
int dtlowf=MS;
int dthighf=YY;
Int4 tsf[7] = { 3, 4, 10 }; /* we want value 10 days,4 hours,
3 minutes */
int tslowf=MI;
int tshighf=DD;
int e;
main()
{
    Datetime dt,dt1,dt2;
    Timespan ts, ts1;
    /* Note: if dt and ts etc. were declared as host variables
inside a ESQLE DECLARE SECTION, you would have to
declare them with a range (Datetime[X:Y] and Timespan[X:Y]);
```

2.3. CONSTRUCTING VALUES OF TYPE DATETIME AND TIMESPAN 27

```
host variables of type Datetime or Timespan always need a range */
char strbuf[50], *dt_str(), *ts_str();
build_dt(&dt,dtlowf,dthighf,dtf);
printf(''dt: %s\n'',dt_str(strbuf,&dt));
build_ts(&ts,tslowf,tshighf,tsf);
printf(''ts: %s\n'',ts_str(strbuf,&ts));
}
build_dt(dt,lowf,highf,dtf)
Datetime *dt;
int lowf, highf;
short dtf[]; /* short! Values come in ascending weight order */
{
    int i, ncomp;
    ncomp = highf-lowf+1; /* number of components */
    for(i=0; i<ncomp; ++i)
        dt->val[i] = dtf[i]; /* store into val[] */
    dtsethighf(dt,highf);
    dtsetlowf(dt,lowf);
    /* dtsetsign(dt,GREGS); */ /* made by dt_check */
    if(e=dt_check(dt))
        err(e);
}

build_ts(ts,lowf,highf,tsf)
Timespan *ts;
int lowf, highf;
Int4 tsf[]; /* Int4! Values come in ascending weight order */
{
    int i, ncomp;
    ncomp = highf-lowf+1;
    for(i=0; i<ncomp; ++i)
        ts->val[i] = tsf[i]; /* store into val[] */
    dtsethighf(ts,highf);
    dtsetlowf(ts,lowf);
    dtsetsign(ts,MINUSS);
    if(e=ts_check(ts))
err(e);
}
err(e)
{
    printf(''%s\n'',tb_errtxt);
    exit(1);
}
```