Date Manipulations in WFL

Relative to every release since 1977 *by Paul H. Kimpel*

At the last UNITE conference in Baltimore, I attended Don Gregory's presentation on virtual files and I/O Handlers, and was listening to his "perverse ideas" for employing them to provide access to special functions in environments which do not support libraries. One of his ideas was for date routines in WFL, to handle typical job flow issues, such as (to use Don's example), "how many days is it until Friday?"

Sitting there, I realized that this is a solved problem, and one that requires a lot less work than designing and writing an I/O Handler. WFL has a decent subroutine mechanism, and while you cannot access libraries or other forms of external intrinsics using it, you can include other WFL source files at compilation time. But can you write a comprehensive set of date routines solely in WFL? The answer is definitely *yes*, using a pair of elegant procedures collectively termed Algorithm 199.

Background

Algorithm 199 was published[1] almost 40 years ago by Robert G. Tantzen of the Air Force Missile Development Center at Holloman Air Force Base, New Mexico. Its name comes from a practice used at the time by the *Communications of the ACM*, the publication in which it appeared, of sequentially numbering each algorithm. It was published in nothing less than glorious, living Algol.

The purpose of Algorithm 199 is to convert between calendar dates in the year/month/day format we traditionally use and a series of sequential day numbers, based on a starting date called the *epoch date*. Traditional dates are termed *Gregorian dates*, after Pope Gregory XIII, who instituted our current calendar system in 1582. The sequential day numbers are termed *Julian day numbers*,[2] presumably after Julius Caesar, who established the prior calendar system in 45 B.C.

Tantzen presented two procedures, `JDAY`, which converts any valid Gregorian date to a Julian day number, and `JDATE`, which converts a Julian day number to its corresponding Gregorian date. He also presented simplified versions of these procedures, named `KDAY` and `KDATE`, which work for dates between 1 March '00 and 31 December '99 in any century.

The WFL date routines presented in this article are based on `KDAY` and `KDATE`. While these simplified routines would normally restrict us from dealing simultaneously with both 20th and 21st century dates, the Y2K problem has, for once, smiled on us. Most century years, while divisible by four, are not leap years, which is what normally prevents `KDAY` and `KDATE` from crossing century boundaries. Because 2000 was a century year divisible by 400, however, it *was* a leap year, and thus these routines can be used for dates from 1 March 1900 to 28 February 2100.[3] This is an adequate range for most business applications.

Once you are able to convert Gregorian dates back and forth to simple day numbers, many date manipulations become trivial. It's easy to determine a date so many days forward or past, to compute the number of days between two dates, or to determine the day of week for a date. The WFL routines presented here provide a basic set of date functions, from which you can build more specialized ones, as a few examples below will illustrate.

The Basic Date Routines

Gregorian dates for the WFL routines described below are passed as binary integer parameters with a format equivalent to decimal `YYYYMMDD`. You can omit the century digits and pass the date as `YYMMDD`, but the routines will assume the century is 1900. Since most parameters are passed to WFL jobs as strings, you can convert a numeric string value to a binary integer value using WFL's `DECIMAL` function, e.g.,

```
INTEGERVALUE:= DECIMAL(STRINGVALUE);
```

If the string value is in `MMDDYY` format, one way that you can convert it to the format required by these routines (assuming you want a date in the 21st century), is this:

```
INTEGERVALUE:= DECIMAL("20" & TAKE(DROP(STRINGVALUE,4),2) & TAKE(STRINGVALUE,2) &
               TAKE(DROP(STRINGVALUE,2),2));
```

You are responsible for formatting the Gregorian date values properly and ensuring they are valid dates. The DATEVALIDATE routine described below can help you with this. The other routines do not validate date parameters. Invalid or out of range date values passed to these routines will generally produce invalid results.

WFL supports subroutines with both call-by-value and call-by-reference parameters, but it does not support typed procedures (functions) which can be used in expressions. Therefore, each of the following routines is written with call-by-value parameters for its input and a call-by-reference parameter for its result. You must declare INTEGER variables to pass and receive values for these parameters.

The basic set of date routines is contained in a standard WFL source file. To use these routines in a WFL job, you must include this source file in the job using a compiler control statement similar to:

```
$$INCLUDE (LIB)WFL/UTIL/DATE/MODULE ON PROD
```

This line must be placed in the job with the other global declarations. Note there are no quotes around the file name in this statement. The file consists of eight subroutines and a set of constant definitions as follows:

DATEKDAY (GREGORIAN, JULDAY);

Converts the integer Gregorian date to a Julian day number. Julian day number 1 (the epoch date) is 1 March 1900. If the Gregorian date has a four-digit year, the year should be in the range 1900-2100. Any other years may produce invalid results. If the year is less than 1900, it is assumed to be a two- or three-digit year based on 1900. Thus a value of 20704 would be interpreted as 4 July 1902, and 1020704 would be interpreted as 4 July 2002. A value of 17760704, which would be 4 July 1776, is invalid.

DATEKDATE (JULDAY, GREGORIAN);

Converts the Julian day number to a Gregorian date in the form YYYYMMDD. The Julian day number should be in the range 1 (1 March 1900) through 73049 (28 February 2100). Other values may produce invalid results.

DATEINCREMENT (GREGORIAN, DAYS, RESULT);

Adds a number of days to a Gregorian date to yield a new Gregorian date in RESULT. The value of DAYS may be negative, allowing you to effectively subtract a number of days from a date.

DATEDIFFERENCE (GREGORIAN1, GREGORIAN2, DAYS);

Determines the number of days between two Gregorian dates. If GREGORIAN1 is earlier than GREGORIAN2, the value of DAYS will be negative.

DATEWEEKDAY (GREGORIAN, DAYOFWEEK);

Determines the day of the week for a Gregorian date. The days are numbered from 0 (Sunday) through 6 (Saturday).

DATEJULIANTOGREGORIAN (JULIAN, GREGORIAN);

Converts a traditional Julian date in the form YYDDD or YYYYDDD to a Gregorian date. As with DATEKDAY, input dates less than 1900 are assumed to be two- or three-digit years based on 1900.

DATEGREGORIANTOJULIAN (GREGORIAN, JULIAN);

Converts a Gregorian date to a traditional Julian date in the form YYDDD or YYYYDDD. The number of digits in the input year is preserved in the resulting Julian date, but the same restrictions on valid year values apply.

DATEVALIDATE (GREGORIAN, VALID);

Tests a Gregorian date for validity. If it is a valid date, the BOOLEAN parameter VALID will be returned with the value TRUE, otherwise it will be FALSE. This routine checks only that the month and day are valid and consistent with the year being a leap year. It does not check for a valid range of centuries.

Constants

The WFL source file defines several constants which you may find useful:

DATEMAXV	21000228	The maximum Gregorian date which the routines will support.
DATEMINV	19000301	The minimum Gregorian date which the routines will support.
DATEYEARBIASV	1900	The century year on which the date routines are based.

DATESUNDAYV	0	The day-of-week values.
DATEMONDAYV	1	
DATETUESDAYV	2	
DATEWEDNESDAYV	3	
DATETHURSDAYV	4	
DATEFRIDAYV	5	
DATESATURDAYV	6	

DATEKDAY and DATEKDATE are used extensively by the other routines listed above, but you will probably not find yourself using them directly very often. Most date calculations can be based on the higher-level routines built from these two fundamental ones.

Some Examples

With this basic set of date routines, we can do a number of interesting things. To address Don's original issue, here is how to calculate the number of days from today until the next Friday. This code assumes that if the current date is a Friday, the result should be zero. The result is left in the variable DIFF.

```
INTEGER TODAY, DOW, DIFF;

TODAY:= DECIMAL(TIMEDATE(YYYYMMDD));      % get today's date
DATEWEEKDAY (TODAY, DOW);                 % DOW = today's day of week
DIFF:= DATEFRIDAYV - DOW;                 % calculate number of days away
IF DIFF < 0 THEN                          % if negative,
  DIFF:= DIFF + 7;                        %   adjust to next week

DISPLAY "THE NUMBER OF DAYS UNTIL FRIDAY IS " & STRING(DIFF,*);
```

Another common problem is to determine the date of the end of the month. The following WFL subroutine will take a Gregorian date and return the date of the last day of the month in which the first date occurs. It works by finding the first of the next month and decrementing that date by one.

```
SUBROUTINE ENDOFMONTH (INTEGER GREGORIAN VALUE, INTEGER EOMDATE);
  BEGIN
  INTEGER
    YY, MM;

  YY:= GREGORIAN DIV 10000;               % determine the input year
  MM:= (GREGORIAN DIV 100) MOD 100 + 1;   % compute the next month
  IF MM > 12 THEN                         % if it's in the next year,
    BEGIN                                 %   adjust accordingly
    MM:= 1;
    YY:= YY + 1;
    END;                                  % construct the date and decrement by 1

  DATEINCREMENT ((YY*100 + MM)*100 + 1, -1, EOMDATE);
  END ENDOFMONTH;
```

As discussed above, the date routines presented here are based on an epoch date in 1900 and use the sensible, but non-traditional YYYYMMDD format. Many operational procedures are based on the more traditional format MMDDYY. The following subroutine takes a Gregorian date in MMDDYY format, deduces the century (it assumes that dates that would be more than five years in the future are in the 20th century), and validates the date. The routine returns the date in YYYYMMDD format, unless it is not valid, in which case it returns zero.

```
SUBROUTINE EDITDATE (INTEGER MMDDYY VALUE, INTEGER YYYYMMDD);
  BEGIN
  INTEGER
    YY, MM, DD, CURRYEAR, RESULT;
  BOOLEAN
    VALID;

  MM:= MMDDYY DIV 10000;
  DD:= (MMDDYY DIV 100) MOD 100;
  YY:= MMDDYY MOD 100;
  CURRYEAR:= DECIMAL(TIMEDATE(YYYYMMDD)) DIV 10000;
  IF YY - CURRYEAR MOD 100 > 5 THEN
    YY:= YY + DATEYEARBIASV
  ELSE
    YY:= YY + DATEYEARBIASV + 100;
```

```
RESULT:= (YY*100 + MM)*100 + DD;
IF RESULT > DATEMAXV THEN
  RESULT:= 0
ELSE
  IF RESULT < DATEMINV THEN
    RESULT:= 0
  ELSE
    BEGIN
    DATEVALIDATE (RESULT, VALID);
    IF NOT VALID THEN
      RESULT:= 0;
    END;

YYYYMMDD:= RESULT;
END EDITDATE;
```

Every installation has its own requirements for handling dates in WFL jobs and will probably want to create a set of custom date routines to handle their most common needs. You can either add such routines to the file of basic date routines, or create additional WFL include files for them. For special cases, you can simply code a custom date routine directly in the source code for the job.

The WFL file of basic date routines, along with a job which illustrates their use (including all of the examples above) can be obtained from the Gregory Publishing web site. You can also download these files from our web site at `http://www.digm.com/Resources/Date`.

Paul Kimpel has been working with MCP-based systems for more than 30 years, starting with the B5500 in the late 1960s. He began his career at Burroughs in 1970 on the then-new B6500. He worked for a number of Burroughs and IBM customers in the late '70s before starting his own business in 1979. Paradigm Corporation specializes in consulting, training, and custom software development for ClearPath MCP systems. Paul's main interests are in the areas of data base and transaction processing system design, web enabling MCP applications, integrating MCP and Microsoft environments, TCP/IP networking, and object oriented programming. His email address is `paul.kimpel@digm.com`.

---

[1] Tantzen, Robert G., "Algorithm 199. Conversions Between Calendar Date and Julian Day Number," *Communications of the ACM*, vol. 8, August 1963, *p*. 444. Also published in *Collected Algorithms from CACM*. This article can be downloaded as PDF from the ACM web site at `http://portal.acm.org/citation.cfm?doid=366707.390020`. You will need an ACM account to access this file.

[2] This use of the term Julian should not to be confused with the `YYDDD` form of date we also call a Julian date.

[3] Although Tantzen described the range of the simplified routines as 1 March '00 through 31 December '99, `KDAY` and `KDATE` are actually good from 1 March '00 of a century through 28 February '00 of the next century. This is a result of the algorithm internally adjusting years to begin on the first of March, which simplifies the handling of leap years. Thus, for the 20th and 21st centuries, the routines will produce valid results from 1 March 1900 through 28 February 2100.

Algorithm 199 – How It Works

Algorithm 199 is a brilliant fudge. If every month had the same number of days, translating between Gregorian dates and Julian day numbers would be easy. Alas, April, June, September, and November each have 30 days, with the rest having 31, except, of course, February, which has 28—unless the year is divisible by 4, in which case it has 29—unless the year is divisible by 100, in which case it reverts to 28—unless the year is divisible by 400, in which case it has 29 again.

The simplified `KDAY` and `KDATE` routines deal with this complexity only within one century. They introduce the concept of a *leap period*, the four years that span from one leap year to the next, consisting of 365×4 + 1 = 1461 days. These routines then shift the start of a year by two months, so that both years and leap periods start on the first of March. This allows the routines to handle the variable number of days in February at the end of a leap period, rather than in the middle.

The cleverest part of these routines is how they translate months and days into day numbers. By shifting the start of the year to the beginning of March and making the month and day numbers zero-relative instead of one-relative, we can build a table like the one at the right showing the relationship between month numbers (m) and the day number (j) at the start of each month.

Month	Days	m	j
March	31	0	0
April	30	1	31
May	31	2	61
June	30	3	92
July	31	4	122
August	31	5	153
September	30	6	184
October	31	7	214
November	30	8	245
December	31	9	275
January	31	10	306
February	28	11	337

The relationship between months and day numbers is almost linear, but not quite. We can create a best fit to the data for this relationship by applying the method of Least Squares.[1] Doing so results in a straight line which can be described by the equation

$$j = 30.6014m + 0.02564$$

This is a nice job of curve fitting, but it doesn't help us much to compute day numbers from month numbers. For that, we need to do the calculations using integer arithmetic. Here comes the fudge. We can make the fractional parts of the coefficients in the equation above very small by multiplying both sides by `5`, thus

$$5j = 153.007m + 0.1282$$

If this were algebra, this last operation would be pointless, but we're not doing algebra, we're doing computing, where things like integer truncation and round-off have significance. Now truncate all the coefficients in this equation to whole integers, thus,

$$5j = 153m + 0$$

Solving again for *j* yields

$$j = (153m + 0) \div 5$$

where the division sign denotes integer division with truncation. If you plug various values of *m* into this equation, as shown in the table to the right, you will find that some of them give the correct number of days, but some do not. If, however (and this is the critical insight that makes the algorithm work), you change the zero in this equation to a `2`, so that

$$j = (153m + 2) \div 5$$

it just so happens that all of the round off and integer truncation issues work in our favor, and for all of the values of *m* this formula yields the correct value for *j* in the table above. It's amazing, but it works, and gives us an easy way to compute the day number at the start of each month.

m	j	(153m+0)÷5	(153m+2)÷5
0	0	0	0
1	31	30	31
2	61	61	61
3	92	91	92
4	122	122	122
5	153	153	153
6	184	183	184
7	214	214	214
8	245	244	245
9	275	275	275
10	306	306	306
11	337	336	337

To compute the day number within a year, we simply need to add the date within the month, *d*, thus,

$$j = (153m + 2) \div 5 + d$$

To expand this formula to work within a century, we need to deal with the problem of leap years and the variable number of days in February. The number of days at the beginning of each March-based year in a century is the number of years times 365 plus the number of leap days thus far in the century, or

$$j = 365y + y \div 4 = (1461y) \div 4$$

Adding these last two equations together, we get the day number (*j*) within a century as the sum of the day number at the start of a year plus the day number for a month and day during the year:

$$j = (1461y) \div 4 + (153m + 2) \div 5 + d$$

which is exactly the formula that the routine `KDAY` uses.

`KDATE` translates from a Julian day number back to a Gregorian date essentially by reversing the sequence of these calculations, using integer and remainder division to carve leap periods, years, months, and days out of the day number.

Tantzen's more general routines, `JDAY` and `JDATE` build on the algorithms of `KDAY` and `KDATE` by adjusting for the exception to the leap year rule that occurs every 100 years, and the exception to the exception that occurs every 400 years. This allows them to correctly convert between Gregorian dates and Julian day numbers across century boundaries. `JDAY` and `JDATE` also adopt the astronomical definition of a Julian day number, which has an epoch date more than 4,000 years B.C.

---

[1] From differential calculus you may recall that the method of Least Squares attempts to find the set of coefficients to a polynomial of the form $y = a_0 + a_1x + a_2x^2 + \ldots + a_mx^m$ that minimizes the sum of the squares of differences between each of the observed data points and the curve of the polynomial. It does this by taking the first derivative of the equations for the sum of the squares of differences, setting the derivatives to zero (to find the minima), and solving the resulting set of simultaneous equations for the coefficients. For a two-dimensional problem (as we have with the relationship between months and Julian day numbers), the method reduces to finding the slope (a_1) and intercept (a_0) of a straight line through the data points.