Trade Program for FastTrack

Contents

Introduction

Trade is a utility program that automates various functions that are useful in the technical analysis and trading of mutual funds. It uses the Investors FastTrack database of mutual fund data. Functions provided by the program include ranking, automated trading, analysis and optimization of automated trading strategies, portfolio FNU generation, generation and analysis of signals, and a powerful expression parser and FNU file generation tool.

The Trade program is run from a command line and requires no interactive user input. It reads commands from one or more text command files and writes results to standard output, FNU, signal, or other files as required.

Trade was written by Ed Gilbert (eyg@prodigy.net). It can be dowloaded from the web page at http://pages.prodigy.net/eyg/trade.

Program Requirements

The Trade program run on any PC that is running Windows95 or Windows NT. It will not run under Windows3.1 or plain MS-DOS because it uses the 32-bit programming API.

The Trade program works with the FastTrack database and will not run if it cannot find the file \ft\ftrack46.dat on the current drive.

Running the program

Using the program typically means creating a command file with a text editor, then entering the filename as an argument on the command line when you run Trade. For example, if you have a command file named slope.ini, you can run it using

```
trade slope.ini
```

It is customary to name command files with a .INI extension, but you can name them anything you like.

You can run multiple command files using a single command as in the example

```
trade slope.ini accu.ini portf.ini
```

Redirecting Screen Output

The output of the Trade program will often fill more than one screen page. To see all of the screen output, you need to use the standard output redirection tool that is built into the Windows command line processing. This tool is the greater-than symbol >.

The > symbol redirects standard output to a file. It creates the file if it does not exist. For example, you could redirect all trade program screen output to a file named list using

```
trade slope.ini >list
```

You can then view the file named list in a text editor or word processor. You can also redirect standard output to your PRN printer as in the example

```
trade slope.ini >prn
```

Online Help

If you run Trade without any command line arguments, it displays a brief summary of its options and arguments. There are a number of help options shown which give more detailed help on a particular command. For example, if you give the command

```
trade -h3
```

it displays help on the Portfolio block commands. Most of the help screens are in the style of an example of a command file. You can use these as a template to create a new command file by redirecting the help output to a file, then editing the file. For example, you can give the command

```
trade -h3 >ira.ini
```

to create the file ira.ini. Then edit ira.ini with a text editor adding your own buy and sell data to create your own Portfolio command file.

Command File Syntax

A command file consists of one or more blocks of commands. Each block begins with a keyword in square brackets, and ends either at the end of the file or when another keyword in square brackets

is read. Here's an example of a command block that runs an Accutrack trading strategy.

```
[AccuFamilyTrade]     ;Trade using Accutrack ranking
AccuFilter = 6, 24
Family = a-mysel
MMarket = fgrxx
Positions = 1         ;Set the number of funds to hold in trading
BuyMinRank = 3        ;Set rank requirement for buy
HoldMaxRank = 16      ;Set rank requirement for sell
HoldMinDays = 30      ;Set the min days to hold a fund before sell
PenaltyDays = 30      ;Set min days to hold to short term Penalty
Penalty = .75         ;Set % penalty if fund sold before HoldMinDays
FnuFile = ACCU3       ;Set the filename prefix for the .FNU file
Signal = T17          ;Set the filename prefix for the .SIG file
InitPosition = FSVLX, 1/2/97 ;Set an an initial trading position
```

Of course, this doesn't explain what all these commands do. These details will be given in another section.

Here are some general rules that apply to all command files.

- Keywords such as AccuFilter, Family, and MMarket are case sensitive and must be entered as shown in this document.

- Filenames, fund symbols, and user-defined expression names are not case sensitive and can be input in any case. You should avoid using names that exactly match Trade keywords in both spelling and case. There is a complete list of Trade keywords in this document.

- Blank lines are ignored and can be used to improve readability.

- Comments begin with a semicolon and continue to the end of the line.

- Dates can be entered using either 2 or 4 digits for the year.

- Space characters are generally not necessary between the various elements of a command. For example, either of the following commands is correct.

  ```
  Penalty=.75
  Penalty = .75
  ```

- Commands within a command block can be placed in any arbitrary sequence after the header, with the exception of the [Expression] block. Expressions are evaluated in the order that they appear in the file, as they would in any programming language.

Command Block Summary

This section gives a brief description of each type of command block.

[Expression]

The Expression commands allow you to combine arithmetic and boolean operators with funds, FNUs, signals, families, and user-defined variables. You can use these expressions to automate the creation of FNU, signal and family files. Any operation that can be performed manually with FastTrack's charting commands can be easily automated. The Expression commands also have a simple report generation capability. Built-in functions include Rsi, Ema, Sma, Max, Min, summation, standard deviation, linear regression, rate-of-change, and many others.

The Expression commands operate on three basic types of data -- scalars, vectors, and signals. Expressions of any of these types can be assigned to user variables, which can then in turn be used in other expressions. Trade also has two complex data types -- families of vectors, and families of scalars. These are discussed in a later section. You should become familiar with the basic types before tackling families.

Scalars and Vectors

Scalars are simply single numbers. For example, the number 33.7 is a scalar.

Vectors are arrays of number, one for each market day. A FastTrack fund symbol is a vector, and FNU files are vectors. When arithmetic operations are applied to vectors, the result is usually a vector as well. For example,

```
ADosc = 100 * NYAV- / (NYAV- + NYDC-)
```

computes an advance-decline oscillator that moves between 0 and 100 and assigns the resulting vector to the variable ADosc. ADosc is a vector because it has a value for each market day in the FastTrack database. ADosc can then be used in an expression as in the example

```
ADmo = Ema(ADosc, 9)
```

which takes the 9-day exponential moving average of every day in the the ADosc vector and assigns that to the variable ADmo, which is also a vector.

Any place in an expression where a vector can be used, an expression returning a vector can also be used. For example, we could write the last 2 lines as a single expression using

```
ADmo = Ema(100 * NYAV- / (NYAV- + NYDC-), 9)
```

The same flexibility also applies to scalars. The second parameter to the Ema() function is a scalar, so any expression or variable that returns a scalar can be used. The two examples below are also equivalent

```
ADmo = Ema(100 * NYAV- / (NYAV- + NYDC-), 5 + 4)
```

and

```
emadays = 9
ADmo = Ema(100 * NYAV- / (NYAV- + NYDC-), emadays)
```

Signals

A signal data type consists of a list of buy dates and a list of sell dates. Signal data types can be created with the Signal() function, and the names of signal files can be used within expressions as if they were signal variables. The Signal() function takes a vector as an argument and returns a signal with buy signals on the dates the vector goes from negative to positive, and sell signals on the dates the vector goes from positive to negative. Here's a simple example

```
Electronics = Signal(Rsi(FSELX, 14) - 50)
```

Here the Rsi() function returns a vector of the 14-day RSI of FSELX. RSI is an indicator that varies between 0 and 100. When 50 is subtracted from Rsi(), the vector returned varies between -50 and 50, and its zero crossings become the buy and sell dates of the signal variable Electronics.

Sometimes you don't want the buy and sell dates to occur at the same thresholds of a vector. The buy and sell parts of a signal variable can be assigned separately by appending .Buy or .Sell to the signal variable. For example, we can the assign buy signals to the Electronics variable on Rsi crossing 45 and sell signals at 55 using

```
Electronics.Buy = Signal(Rsi(FSELX, 14) - 45)
Electronics.Sell = Signal(Rsi(FSELX, 14) - 55)
```

The .Buy and .Sell qualifiers are only used with signal variables on the left hand side of an assignment (=).

Sometimes you want to combine two signals using boolean logic. Trade expressions have And, Or, and Not operators for this purpose. For example, if you have two signals named t17 and djusp, you can combine using

```
newsig = t17 And djusp
```

In this example, newsig will have buy dates when both t17 and djusp are in buy mode, and sell dates when both are in sell. If

you are a user of Brian Stock's FastTools, this is the same as FastTools confirm with the "1" suffix. Using the And, Or and Not operators with the .Buy and .Sell qualifiers, you can create any logical combination of any number of signals.

You can use the And, Or, and Not operators with vectors as well as signals. When you Or two vectors, the result is a vector with a value of 0.5 for each day that either operand is positive, and a value of -0.5 for each day that both are negative. And and Not behave similarly with vectors.

Trade has a function which converts a signal to a vector. The Vector() function takes a signal argument and returns a vector with a value of 0.5 for each day that the input signal is buy, and a value of -0.5 for each day that the signal is sell. You can use this feature to make a consensus signal out of a number of individual signals. An example of this is given later.

At some point you will want to write your vectors as FNU files, or your signals as .SIG files. This is done with the WriteFile() command. The first argument to Writefile() is a vector or signal expression, and the second argument is the filename (prefix only, Trade supplies the .sig or .fnu as necessary, and puts the file in the proper FastTrack directory). For example

```
WriteFile(Sum(nyav- - nydc-) + 500, nyad)
```

writes the FNU file \ft\nyad.fnu as the daily summation of the NY advancing - declining issues with the constant 500 added to each days value. You can write a signal file in the same way as in the example

```
newsig = t17 And djusp
WriteFile(newsig, usp17)
```

which writes the file \ft\sig\usp17.sig from the signal dates in the variable newsig. WriteFile() knows what type of expression the first argument is, and writes the appropriate file for that type.

Print

Trade has a simple means of displaying the last few values of an expression at standard output. The command Print() takes an expression and descriptive text enclosed in quotes, and formats it at standard output. For example

```
ADmo = Ema(100 * NYAV- / (NYAV- + NYDC-), 9)
Print(ADmo, "Don Bell's ADMO")
```

produces the following output

	07/23/97	07/22/97	07/21/97	07/18/97	07/17/97

```
    Don Bell's ADMO     53.41     51.82     46.95     50.42     55.55
```

If you need more space for the descriptive text, you can embed newlines in it to continue on the next line. When Print sees a \n sequence in the text string, it outputs the following part of the string on the next line. For example

```
    ADmo = Ema(100 * NYAV- / (NYAV- + NYDC-), 9)
    Print(ADmo, "Don Bell's ADMO\nBuy 62, Sell 49")
```

produces the output

```
                    07/23/97  07/22/97  07/21/97  07/18/97  07/17/97
                    --------  --------  --------  --------  --------
    Don Bell's ADMO     53.41     51.82     46.95     50.42     55.55
    Buy 63, Sell 49
```

One of the most common uses of the Print() command is to make a table of the outputs of several vectors. You can give several vector,"string" pairs to Print to get a more compact output in tablular format, with column headings shared by all the vectors. For example

```
    ;Don Bell's Composite Breadth Oscillator for NYSE
    nyadbr = 100 * (nyav- - nydc-) / (nyav- + nydc-)
    nyudbr = 100 * (nyuv- - nydv-) / (nyuv- + nydv-)
    nyhlbr = 100 * (nynh- - nynl-) / (nynh- + nynl-)
    nybr = (nyadbr + nyudbr + nyhlbr) / 3
    nybr = Ema(nybr, 5)
    Print(nyadbr, "NYAD Breadth Osc",
          nyudbr, "NYUD Breadth Osc",
          nyhlbr, "NYHL Breadth Osc",
          nybr, "Composite Osc")
```

produces the output

```
                     09/18/97  09/17/97  09/16/97  09/15/97  09/12/97
                     --------  --------  --------  --------  --------
    NYAD Breadth Osc     18.82      2.46     49.43      8.92     45.30
    NYUD Breadth Osc     27.67     -0.02     55.60     -1.91     50.04
    NYHL Breadth Osc     92.57     91.56     93.01     91.22     83.11
       Composite Osc     42.13     40.02     44.36     33.53     33.93
```

You can also use the Print() command with scalars or signals. These produce only a single line of output. For example,

```
    Print(Last(fselx), "Latest fselx close is")
    Print(t17, "Dennis Meyers' T17")
```

produces the output

```
    Latest fselx close is  45.19

    Dennis Meyers' T17  (Last: Buy, 05/01/97)
```

If, Else, EndIf

These statements conditionally execute a group of Expression commands based on a numeric value or the state of a signal. The If statement tests a condition. If it is True, then the statements following the If statement up to a corresponding Else or EndIf are executed. If the condition is false, then the immediately following statements are not executed until a corresponding Else or EndIf is found. Used with a vector or scalar, the condition is True if the value is zero or positive, and False if it is negative. Only the most recent value of a vector is tested. Used with a signal, the condition is True if the current state of the signal is Buy, and False if it is Sell.

The Else statement is optional. When used after an If, the commands following the Else are executed if the condition tested by the corresponding If is false.

Syntax

```
If (condition)
   statement
   statement
     ...
Else
   statement
   statement
     ...
EndIf
```

If statements can be nested to any depth. The following is a simple example of nested If statements. (Indenting statements inside an If or Else block is not required, but improves the readability).

```
If (fselx - Ema(fselx, 39))
   If (t117)
      Print("T117 is Buy, and fselx is above its 39d Ema")
   Else
      Print("T117 is Sell, and fselx is above its 39d Ema")
   EndIf
Else
   Print("fselx is below its 39d Ema")
EndIf
```

Note that since If (condition) only tests the last value of a vector, it cannot be used to make conditional assignments to every price in a vector. For example, you cannot take the absolute value of every price in a vector using

```
If (vector)
   absval = vector
Else
```

```
        absval = -vector
     EndIf
```

because the If statement only tests the last value of vector, but the assignments to absval assign all values as either vector or -vector. Because If makes only one test, its most useful application is with Print in providing a printed output that unravels some complex decision tree based on daily computed indicators. If you are trying to do something that does require that you test each price of a vector and make some conditional assignment for each price, you can usually accomplish this with the Sgn function. For example, you can compute the absolute value of a vector using

```
     positive = Sgn(vector) + 0.5
     absval = vector * positive - vector * (1 - positive)
```

Examples

The next example computes the accutrack 5, 35 of fsesx paired with fsvlx and writes the signal file \ft\sig\esvl.sig based on this pairing.

```
    AccuEsvl = Accu(fsesx, fsvlx, 5, 35)
    WriteFile(Signal(AccuEsvl), esvl)
```

The next example creates the signal file \ft\sig\djsig.sig based on DJ-30 crossing its 39-day Ema.

```
    djfiltered = Ema(dj-30, 39)
    WriteFile(Signal(dj-30 - djfiltered), djsig)
```

The next example creates Bollinger bands (20 days, 2 standard deviations) for fselx, and writes the signal file \ft\sig\bbsig.sig. Buys occur when fselx crosses its lower band in a positive direction. Sells occur when fselx crosses its upper band in a negative direction.

```
    days = 20
    upperband = Sma(fselx, days) + 2 * Stdev(fselx, days)
    lowerband = Sma(fselx, days) - 2 * Stdev(fselx, days)
    bb.Buy = Signal(fselx - lowerband)
    bb.Sell = Signal(fselx - upperband)
    WriteFile(bb, bbsig)
```

The next example displays the recent MACD(30, 15, 10) of vfinx.

```
    diff = Ema(vfinx, 15) - Ema(vfinx, 30)
    macd = diff - Emaz(diff, 10)
    Print(50 + 100 * macd, "MACD(VFINX, 30, 15, 10)")
```

The next example displays the recent 20-day stochastic of dj-30.

```
    DJstr = 100 * (dj-30 - Min(dj-30, 20)) /
                  (Max(dj-30, 20) - Min(dj-30, 20))
    Print(DJstr, "20-day stochastic of DJ-30")
```

The following example automates the bond trading strategy described in FastTrack commentary #7000. This scheme uses the unadjusted (without dividends) prices of FBNDX and FAGIX to create an accutrack based timing signal.

```
    ;Get the raw fund prices with dividends removed.
    fund_raw = Raw(fbndx)
    index_raw = Raw(fagix)

    ;Compute Accutrack 20,20 of fund vs index.
    accuBnd = Accu(fund_raw, index_raw, 20, 20)

    ;Create buy and sell signals on accutrack zero crossings.
    BondSig = Signal(accuBnd)
    WriteFile(BondSig, bsig)
```

The following example creates the consensus signal file \ft\sig\combo.sig that is a buy when any 2 of the 3 input signals are a buy, and a sell when any 1 of the 3 are a sell. The 4th line creates a vector of numeric sums of all the signals that varies from 0 on days when they are all sell to 3 on days when they are all buy. You can extend this concept to any number of signals by changing the constants on the first 3 lines and substituting your own list of signals.

```
    buycount = 2
    sellcount = 1
    totalcount = 3
    sigsum = Vector(djusp) + Vector(t17) + Vector(t87) + Totalcount / 2

    ;Create a vector that goes positive when sigsum increases.
    bstate1 = (sigsum - Shift(sigsum, 1) - 0.5)

    ;Create a vector that is positive when sigsum >= buycount.
    bstate2 = sigsum - buycount + 0.5

    ;Create a vector that goes positive when sigsum decreases.
    sstate1 = Shift(sigsum, 1) - sigsum + 0.5

    ;Create a vector that is positive
    ;when #sells >= sellcount.
    sstate2 = totalcount - sellcount - sigsum + 0.5

    combo_sig.Buy = Signal(bstate1 And bstate2)
    combo_sig.Sell = Not Signal(sstate1 And sstate2)
    WriteFile(combo_sig, combo)
```

You can add weightings to each signal in a consensus signal so that each signal contributes a different weight to the result. In the example below, A, B, C, and D are signals, each with

relative weights wa, wb, wc, wd. The buy and sell thresholds buy_pcnt and sell_pcnt are set as percentages, so the buy_pcnt of 70 means that the newsig will have a buy when the weighted sum of the signals is 70% towards the all-buy weight, and the sell_pcnt of 60 means that the newsig will have a sell when the weighted sum is 60% towards the all-sell weight.

```
    wa = 35         ;set relative weight of signal a
    wb = 25         ;set relative weight of signal b
    wc = 20         ;set relative weight of signal c
    wd = 15         ;set relative weight of signal d
    buy_pcnt = 70   ;set buy percentage threshold
    sell_pcnt = 60  ;set sell percentage threshold
    sum = wa * (Vector(a) + 0.5) +   ;Sum all the vectors
          wb * (Vector(b) + 0.5) +
          wc * (Vector(c) + 0.5) +
          wd * (Vector(d) + 0.5)
    sum = 100 * sum / (wa + wb + wc + wd) ;Normalize range to 0 to 100

    ;Create a vector that goes positive when sum increases.
    bstate1 = sum - Shift(sum, 1)

    ;Create a vector that is positive when sum >= buy_pcnt
    bstate2 = sum - buy_pcnt

    ;Create a vector that goes positive when sum decreases.
    sstate1 = Shift(sum, 1) - sum

    ;Create a vector that is positive when sum <= sell_pcnt.
    sstate2 = sell_pcnt - sum

    newsig.Buy = Signal(bstate1 And bstate2)
    newsig.Sell = Not Signal(sstate1 And sstate2)
    WriteFile(newsig, comb)
```

The following summary is a complete list of Expression functions and operators.

Functions which return vectors:

Roc(vector, Ndays)
Returns a vector which is the rate of change of the input vector over Ndays. For each day, the rate of change is calculated as

Roc = (TodaysNav - NavNdaysAgo) / NavNdaysAgo.

Ema(vector, Ndays)
Emaz(vector, Ndays)
Both functions return a vector which is the exponential moving average of the input vector. For each day, the Ema is

```
   K = 2 / (Ndays + 1)
   Ema = TodaysNav * K + Ema * (1 - K)
```

The difference between Ema and Emaz is in the starting value of the filter. Ema uses the first value of the input vector as the initial value of the filter. Emaz uses zero as the initial value. These functions both converge to the same result after a few Ndays time constants. The Ema function is more appropriate for most funds because funds have a starting price which is offset from zero. The Emaz function is more appropriate for oscillators or other indicators which may swing above or below zero.

The Ndays argument can be either a scalar or a vector. Using a vector allows variable length Emas to be computed.

Sma(vector, Ndays)

Returns a vector which is the simple moving average of the input vector over Ndays. For each day, the Sma is the sum of the values of the previous Ndays divided by Ndays.

Rsi(vector, Ndays)

Returns a vector which is the Welles Wilder Rsi of the input vector. Rsi is computed each day as follows:

```
   K = 1 / Ndays
   price_chg = TodaysNav - YesterdaysNav
   pos_chg = GreaterOf(price_chg, 0)
   up_avg = pos_chg * K + up_avg * (1 - K)
   neg_chg = LesserOf(price_chg, 0)
   down_avg = neg_chg * K + down_avg * (1 - K)
   Rsi = 100 * up_avg / (up_avg + down_avg)
```

Accu(fundVector, indexVector, filter1, filter2)

Returns a vector which is the Accutrack indicator of a fund paired with an index. The scaling of the values is the same as used by FastTools. Buy and sell signals computed at the zero crossings should match the signal dates created by the FastTrack charting program. The Ema filters are initialized to average values over the number of days given by filter1 in order to match FastTrack's signal dates near the beginning of the database. With the exception of these initial values, the Accu() function is computed as follows:

```
   delta = Ema(Roc(fundVector, 1), filter2) -
             Ema(Roc(indexVector, 1), filter2)
   accu = Ema(delta, filter1) * 26500
```

SignalPairTrade(fundVector, indexVector, signal)
SignalPairTrade(fundVector, indexVector, signal, delay)

Returns a vector which is the result of trading between two vectors using a signal. An optional delay arguement sets the buy and sell delay between the signal dates and trade

dates. If delay is not given, a delay of 1 day is used.

Max(vector, Ndays)
Returns a vector which is the maximum value of the input vector found over the last Ndays.

Min(vector, Ndays)
Returns a vector which is the minimum value of the input vector found over the last Ndays.

Sum(vector)
Returns a vector which is the cumulative summation of every value in the input vector.

FamAvg(family)
If the input is a vector family, FamAvg returns a vector with a daily return that is the average of the daily returns of all of the vectors in the family. If the input is a scalar family, FamAvg returns a scalar that is the average value of the input scalars.

FamAD(family)
Returns a vector which is the a count of the number of advancing funds in the family minus the number of declining funds in the family.

Raw(symbol)
Returns a vector which is the unadjusted raw NAVs of a FastTrack database symbol.

Shift(vector, Ndays)
Returns a vector which is the input vector shifted by Ndays.
If Ndays is positive, then output[i] = vector[i - Ndays].
If Ndays is negative, then output[i] = vector[i + Ndays].

Lreg(vector, Ndays)
Returns a vector in which each value is the point on the linear regression line of the last Ndays of the input vector.

Stdev(vector, days)
Returns a vector which is the standard deviation of the previous Ndays.

Sgn(vector)
Returns a vector which gives the sign of the values of the input vector: 0.5 if zero or positive, -0.5 if negative.

Vector(signal)
Returns a vector which has a value of 0.5 when the input signal is in a buy state, and a value of -0.5 when it is in a sell state.

Not(vector)

Returns a vector which has a value of -0.5 when the input vector is positive or zero, and a value of 0.5 when the input vector is negative.

Log(vector)
Log(scalar)
Returns a vector or scalar which is the base 10 logarithm of the input price(s).

Pow(x, y)
Returns a vector or scalar which is x raised to the y power. If x, y or both are vectors, a vector is returned. If both are scalars, a scalar is returned.

Days(signal)
Returns a vector which is the number of days since the most recent change of state of the input signal.

Irate(pcnt)
Returns a vector with a constant annual percentage interest rate.

Warp(vector, beta, offset)
Returns a vector whose exponential gain is multiplied by beta and incremented by an offset. The offset argument is given as an equivalent annual percentage. If vector[0] is the price on the first day and vector[d] the price on day numbered d, then the expression for the output vector on day d (outvector[d]) is

```
MKT_DAYS_PER_YEAR = 254
i = (1.0 + offset / 100.0) ^ (1.0 / MKT_DAYS_PER_YEAR)
irate_mult = i ^ d
gain_mult = (vector[d] / vector[0]) ^ beta
outvector[d] = vector[0] x irate_mult x gain_mult
```

Functions which return signals:

Signal(vector)
Returns a signal with Buys on positive going dates and Sells on negative going dates.

Not(signal)
Returns a signal which is the logical negation of the input signal. It has sells when the input signal has buys, and buys when the input signal has sells.

Edit(signal, day or date, Buy or Sell)
Returns a signal which is the input signal with a forced Buy or Sell on a given market day or date. The day can be given either as a market day number or as a date in mm/dd/yy or mm/dd/yyyy format. Market days are numbered starting with 0 for 9/1/88. The market day of the latest day in the FT database can always be computed by summing a vector of 1's

as in the example

```
LastMarketDay = Last(Sum(Irate(0) - Irate(0) + 1)) - 1
```

Edit(signal, start_date .. end_date, Buy or Sell)
Edit(signal, start_day .. end_day, Buy or Sell)
Returns a signal which is the input signal with forced Buys or Sells on a range of market days or dates.

Edit(vector, day or date, value)
Returns a copy of the input vector with value assigned on the given market day or date.

Edit(vector, start_date .. end_date, value)
Edit(vector, start_day .. end_day, value)
Returns a copy of the input vector with value assigned on the given range of market days or dates.

Stretch(signal, Ndays)
Returns a signal which is the input signal with at least Ndays market days between signal changes.

ZigZag(vector, Npcnt)
Returns a signal in which the buys and sells are local maxima and minima of the input vector. Each buy or sell represents at least an Npcnt percent change from the previous signal. ZigZag can be used to identify optimum buy and sell dates in analyzing the past history of a fund, but it cannot be used as a realtime trading tool because it does not decide on a buy or sell date until the price has already reversed and moved at least Npcnt percent in the opposite direction.

Functions which return scalars:

First(vector)
Returns the value on the first day of the input vector.

Last(vector)
Returns the value on the last day of the input vector.

Miscellaneous Functions:

WriteFile(expression, fname)
Write an FNU or SIG file. If expression is a vector, the file \ft\fname.fnu is written. If expression is a signal, the file \ft\sig\fname.sig is written, where fname is the filename prefix argument given to WriteFile().

WriteFile(expression, fname, "description")
Write an FNU or SIG file as described above. The optional description string is written in the header of the file.

Print("text")

Print(expression, "text", ...)

Print either a text string, or one or more expression, "text" pairs to standard output. If expression is a scalar, its value is printed. If expression is a signal, its most recent signal change and date are printed. If expression is a vector, the 5 most recent values are printed in columns with date headings. Newlines can be embedded in "text" using the character sequence \n.

Within the "text" string, several macros are expanded. These macros are recognized regardless of case.

$Sym	prints the symbol of a vector.
$Des	prints the FT description of a vector.
$Cdate	prints the current date in mm/dd/yyyy format
$Mdate	prints the most recent market date in mm/dd/yy format
$Tdate	prints the current time and date

Operators:

Trade expressions use the following operators, which are listed in the order of their precedence, from highest to lowest.

()	grouping
-	unary negation
* /	multiplication, division
+ -	addition, subtraction
And Or Not	logical operations on vectors and signals
=	assignment

Variables:

Expression variables can be up to 40 characters in length. Case is not significant, but care should be take that they do not exactly match other Trade keywords in both case and spelling. Variables assume the expression type that they were assigned from (scalar, vector, or signal).

When a name is parsed in an Expression, Trade first compares the name to the user-defined variables, then FNU files, then FastTrack symbols, then signal file names, then family file names in the \ft\userdef directory, and finally family file names in the \ft\ftdef directory. The first match that it finds gets used in the expression. For example, if you give the command

```
fselx = 5
```

then you can no longer use fselx as a vector within that Expression block because it has been re-defined a scalar. Similarly, if you have signal files named btttx.sig and mirat.sig and you give the command

```
mysig.Buy = btttx And mirat
```

you will get an error because btttx is also the symbol of a fund, and you cannot directly And a fund with a signal.

The best solution is to keep all names of fnu, signals, and families unique, but if you must reference a signal or family file whose name conflicts with something before it in the search order, you can refer to the file with its filename extension. In the example above with btttx, you can reference the signal file btttx using

```
mysig.Buy = btttx.sig And mirat
```

When you reference btttx.sig in this way, the symbol btttx gets put into Trade's symbol table as a signal variable, and all subsequent references to btttx will get the signal, not the fund named btttx.

The same tactic works with family names. For example, if for some reason you had a family file named fselx, you would not be able to compute its advance-decline vector with

```
adl = FamAD(fselx)
```

because the fund named fselx hides the family, however you could do it with

```
adl = FamAD(fselx.fam)
```

and subsequently this would work correctly

```
avg = FamAvg(fselx)
```

A disadvantage of this technique is that the fund named fselx is now hidden. You can restore the visibility of the fund fselx by deleting the family symbol fselx from Trade's symbol table using the Delete() function. The primary purpose of the Delete() function is to free up memory used by large families after they are no longer needed, but it can also be used to remove a symbol that hides access to another name. Thus is you gave the commands

```
btsig = btttx.sig
Delete(btttx)
```

you then have access to the signal named btttx through btsig, and access to the fund named btttx through its own name.

Signal Logic

When two signals are combined with the And operator as in the example

```
foo = A And B
```

the result signal foo signals a Buy when both A and B are in a Buy state, and signals a Sell when both are in a Sell state.

When two signals are combined with the Or operator as in the example

```
foo = A Or B
```

the result signal foo transitions to a Buy when either A or B transitions to a Buy, and similarly it transitions to a Sell when either transitions to a Sell.

The Buy and Sell dates of two independent signals can be combined using the .Buy and .Sell qualifiers as in the example

```
foo.Buy = A
foo.Sell = B
```

However, if A and B both transition in opposite directions on the same day, there will not be a change in foo on that day.

Vector Start Dates

Every vector has an associated start date which is the first date for which the vector has data. While many of the FT funds have start dates of 9/1/88, some did not exist on that date and have later start dates. For example, oaksx has a start date of 11/2/95. When you write an expression using 2 vectors, the operation is only performed on dates for which both vectors have valid data. This means that the start date of the result is the later of the start dates of the two input vectors. For example, if you compute the relative strength of oaksx to rut-i using

```
RelStr = oaksx / rut-i
```

then RelStr is a vector with a start date of 11/2/95, even though Russell 200 index rut-i has a start date of 9/1/88. If you create an fnu file of RelStr, it's data will start on 11/2/95.

The start date is an attribute that is somewhat hidden, but there is a way to compute how many market days have elapsed since a vector's start date. Create a vector whose data on each date is the value 1 and with the same start date as the vector in question. Use the Sum function to create a vector of cumulative summations, and then use the Last function to most recent value of the summation. For example, the number of market days of data for the fund oaksx can be computed with

```
NumDays = Last(Sum(oaksx - oaksx + 1))
```

Changing Vector Start Dates

Although you shouldn't need to do this, there are ways to change

the start date of a vector if it's needed for some unusual application. A vector's start date can be advanced forward as in the following example which moves the start date of oaksx forward by 25 days:

```
oaksx_clipped = oaksx + Shift(oaksx, 25) - Shift(oaksx, 25)
```

If you use the Edit function with a date that is earlier than the start date, the start date of the returned vector is the edit date. For example

```
OakEdited = Edit(oaksx, 9/1/88, 10)
```

creates the vector variable OakEdited with a start date of 9/1/88. Its values are 10 from the start until 11/2/95, from which point its values are the same as oaksx.

Families

Trade has three family data types which allow expressions to operate on entire families. Almost all of the previously described functions that operate on vectors, scalars and signals, including Print() and WriteFile(), can be used with the family types. There are also some specialized functions for family creation, sorting, selection (creating subsets of families), and report generation that operate exclusively with the family types. Family data types greatly expand the capabilities of Trade expressions by allowing a single expression to be applied to every member of a family.

Vector Family

A "vector family" is an ordered list of vectors. A FastTrack family filename can be used as a vector family. For example, you can compute the 66-day growth rate of every fund in the VALUE family using the expression

```
roc66 = 100 * Roc(value, 66)
```

The variable roc66 in this example becomes a vector family data type which holds the growth rate percentages.

You can also create family vectors inline using the Family() function. The following example creates a family vector of nine junk bonds.

```
JunkBonds = Family(sthyx, jahyx, fhypx, fahyx, sphix, fagix,
                   prhyx, nthex, vwehx)
```

Trade does not place any limits on the number of vectors you can put in a family. You can also place Trade variable names into families.

Scalar Family

A "scalar family" is an ordered list of scalars. A scalar family is returned by the First() and Last() functions when given vector families as input arguments. Thus the example

```
latest_roc66 = 100 * Last(Roc(value, 66))
```

creates a family of scalars containing the most recent growth rate percentages of the value family.

You can also create scalar families inline using the Family() function, as with the example

```
EmaDays = Family(11, 13, 15.5, 21.22)
```

and you can include Trade scalar variable names that you have previously defined into scalar families.

You cannot mix scalars and vectors within a family. It would be an error to try to declare something like

```
nonsense = Family(fselx, 5.5, fsvlx)
```

Signal Family

A "signal family" is an ordered list of signals. Any function that can return a signal will return a signal family if any of its input arguments are families. For example

```
fam = select
sig = Signal(Ema(fam, 19) - Ema(fam, 39))
```

creates a family of signals for the Fidelity select family.

Unlike vector and scalar families, signal families cannot be created inline with the Family() function.

Family Functions

The Print() function is used to output the last few values of each member of a family to standard output, just as it is with simple data types. With families there is an additional need to identify the individual members in each line of the Print() output. Every member of a family has both a Symbol and a Description, and these attributes can be referenced in the descriptive titles that are output with each Print() line. To reference these attributes in the title strings, use the following macros:

```
$sym    is replaced by the family member's Symbol.
$des    is replaced by the family member's Description.
```

For example, if we give the commands

```
    roc66 = 100 * Roc(value, 66)
    Print(roc66, "$sym $des")
```

then the first few lines of output look like

```
                              08/03/98  07/31/98  07/30/98  07/29/98
07/28/98
                              --------  --------  --------  --------  --
------
AVAIX Accessor Value & Income     -4.32     -2.88     -1.31     -4.47
-5.50
AVLBX Aim Value-B                  5.19      7.24      9.37      5.28
4.64
AVLFX Aim Value-A                  5.43      7.47      9.59      5.48
4.85
BVALX Babson Value                -8.42     -6.95     -5.13     -7.70
-8.33
CGRWX Oppenhmr Disciplined Va     -5.38     -4.11     -1.88     -5.01
-5.26
DAGVX Dreyfus Gr & Val: Agg V     -2.60     -1.84      0.36     -3.19
-4.24
DCVIX Dreyfus Premier Core Va     -5.97     -4.90     -3.56     -6.57
-7.47
```

A useful function is the ability to sort a family. The Sort() function returns a copy of the input family with the members sorted by their most recent values in descending order. Thus the commands

```
    roc66 = 100 * Roc(value, 66)
    Print(Sort(roc66), "$sym $des")
```

produce these first few lines of output

```
                              08/03/98  07/31/98  07/30/98  07/29/98
07/28/98
                              --------  --------  --------  --------  --
------
LMVTX LeggMasn Value               6.22      7.38      9.64      4.50
4.70
IAAPX Iai Value                    5.68      7.24      8.56      5.50
5.34
AVLFX Aim Value-A                  5.43      7.47      9.59      5.48
4.85
AVLBX Aim Value-B                  5.19      7.24      9.37      5.28
4.64
RLVAX Reserve LargeCap Value       3.71      4.80      6.90      3.78
3.21
VALLX ValuLine Leveraged Grow      3.23      5.52      8.51      3.53
3.39
```

```
VALBX ValuLine US Gov Securit      2.71      2.34      2.43      1.79
1.97
```

To sort in ascending order, simply negate the family before and after the sorting process as in

```
Ascending = -Sort(-roc66)
```

Table Printing with FamPrint()

A very useful and powerful feature is the ability to compute several indicators from a base family, and then print a multi-column table of these indicators. This is most easily explained with an example. Suppose we want to print a table for the VALUE family with the following columns:

- Column 1: The fund symbol and description.
- Column 2: The sorted Accutrack(11, 44) value against a 0% index, in descending order.
- Column 3: The 66-day average growth rate.
- Column 4: The 21-day Rsi.
- Column 5: The 254-day ulcer index.

We will compute the ulcer index separately, since all the other columns are one-line expressions which can be conveniently computed right inside the FamPrint() function.

```
[Expression]
fam = value
UIdays = 254
peak = Max(fam, UIdays)
cdd = (peak - fam) / peak
ui = 100 * Pow(Sma(cdd * cdd, UIdays), .5)
```

Now we're ready for the FamPrint() function. It's format is

```
FamPrint(family1, "title1", family2, "title2", ...)
```

It's a little different than the Print() function. Up to 7 families and their column titles can be specified. The columns are wide enough for 8 digits of printing, so the titles have to be short and are truncated to 8 characters. Since FamPrint() is dedicated to printing family tables, there is no need to specify the symbol and description for the leftmost column -- it is printed automatically. Here's the command for our example.

```
FamPrint(
    Sort(Accu(fam, Irate(0), 11, 44)), "AT11/44",
    100 * Roc(fam, 66), "Roc(66)",
    Rsi(fam, 21), "Rsi(21)",
    ui, "UI(254)")
```

When run on 8/3/98, it produces the output below. Only a few lines are shown here of the complete output from this large family.

	AT11/44	Roc(66)	Rsi(21)	UI(254)	
LMVTX LeggMasn Value	23.61	5.47	48.29	3.83	
AVLFX Aim Value-A	20.15	5.43	44.99	3.58	
AVLBX Aim Value-B	19.32	5.19	44.77	3.66	
GABVX Gabelli Value	18.92	2.45	47.43	1.95	
RLVAX Reserve LargeCap Value Equity	8.93	3.71	44.98	2.59	
VALLX ValuLine Leveraged Growth Inves	8.65	3.23	43.53	4.79	
VAGIX ValuLine Aggressive Income	6.17	0.81	70.17	0.60	
VALIX ValuLine Income	5.89	1.40	43.09	4.33	
VALBX ValuLine US Gov Securities	5.48	2.71	62.98	0.58	
PLCVX PBHG LargeCap Value	5.06	1.94	44.28	2.57	

The first family column (AT11/44 in the example) is the "key" for the table. It determines the order in which the members of each family are printed. In the example, LMVTX was the first fund in the column 1 family. It was not the first fund in families of the other columns. Those families were not re-arranged by Sort() and retained their original ordering from the VALUE family, yet the FamPrint() was able to locate their members and print them on the correct lines. If FamPrint() cannot find a particular member of a column family, it prints a field of dashes "-----" for that entry.

Selection

The Select() function takes an input family and returns a family of only those input members whose most recent values meet some selection criteria. If Ann is a family, then

```
Select(Ann >= 20)
```

returns a family of all the members of Ann with last values of at least 20.

The general format is

```
    Select(family test limit)
```

where test can be one of

```
    <=        less than or equal to
    <         less than
    >=        greater than or equal to
    >         greater than
```

Family Set Operations

Trade has family operators which perform the three primary set operations (union, intersection, and complement) using the names of the family members.

Intersection is performed by the FamAnd operator. The expression

```
    family1 FamAnd family2
```

returns a family with members of family1 that are also contained in family2. Note that Trade allows mixing a scalar family with a vector family as the two arguments because the returned family can only be a subset of family1.

The set complement operation is performed by the FamAndNot operator. The expression

```
    family1 FamAndNot family2
```

returns a family with the members of family1 that are not contained in family2. For example, you can create a family variable that has all the precious metal funds removed from the all-eq family using

```
    AllEqNoMetals = all-eq FamAndNot precious
```

As another example, you can create a subset of the Fidelity select family by removing some funds specified inline using

```
    MySelects = select FamAndNot Family(fsagx, fsesx, fdpmx)
```

The set union operation is performed by the FamOr operator. The expression

```
    family1 FamOr family2
```

returns a family that has all the members of family1 plus those members of family2 that are not in family1. For example, you can combine two families using

```
    SmallCaps = cap-micr FamOr cap-smal
```

Keep in mind that the set operations do not look at the data values contained in the families, but the family returned by the operation has the data values of the input members. For the FamAnd and FamAndNot operators, the returned values all come from the first family argument, but with FamOr, the returned family will have members from both inputs. This is fine if all you are going to do is write a family file with the result, but if you are going to use the result in some subsequent calculations, you will want the values in the result to represent the same type of quantity (have the same units).

Suppose, for example, that you have started with a large family like B-FI-RET, and through some expressions you have created a small family subset named UI which has ulcer indices for its values. You would like to take the funds in UI and compute some other indices, but you need a family with the funds that are in UI but you want the values to be the NAVs of the funds from the original B-FI-RET. This can be obtained using

```
family = b-fi-ret FamAnd ui
```

You can use FamAnd in combination with the FamPos() function to pick the top funds from a family using some computed indicator. FamPos(family) returns a scalar family with the positions of the input members as the scalar values. For example, if you have computed the UPI of a B-FI-RET, you can create a family of the top 20 UPI funds using

```
top20pos = Select(FamPos(upi) <= 20)
```

You can then make a family of the orignal NAVs of the top20 funds using

```
top20 = b-fi-ret FamAnd top20pos
```

and you can make a family of their UPI values using

```
top20upi = upi FamAnd top20pos
```

All three family set operators have left associativity and equal precedence, thus the expression

```
a = b FamAnd c FamOr d FamAndNot e FamOr f
```

is equivalent to

```
a = (((b FamAnd c) FamOr d) FamAndNot e) FamOr f
```

Unique Functions of Scalar Families

The following functions are somewhat unique in that they are not normally used with scalar arguments, but they perform useful functions with scalar family arguments.

First(family)
     Returns the first member of the input scalar family.

Last(family)
     Returns the last member of the input scalar family.

Min(family)
     Returns the member of the input scalar family with the
     lowest value. Note that there is no Ndays argument when
     used with a scalar family.

Max(family)
     Returns the member of the input scalar family with the
     highest value. Note that there is no Ndays argument when
     used with a scalar family.

Other Family Functions

FamPos(family)
     Returns a scalar family with all the members of the input
     family, and the member positions as the scalar values. The
     input can be a scalar family or a vector family.

GetMember(family, name)
     Finds and returns the family member with the given name.
     GetMember returns a vector if family is a vector family, or
     a scalar if family is scalar family.

FamSum(family)
     If the input is a vector family, FamSum returns a vector
     which is the sum of the daily values of each family member.
     If the input is a scalar family, FamSum returns a scalar
     which is the sum of the values of each family member. If
     input is a scalar family, FamSum returns a scalar sum. If
     the input is a vector family, FamSum returns a vector where
     the value on each day is the sum of the family values on
     that day.

FamCast(family, signaturefam)
     Returns a copy of the first family argument, but with the
     "signature" of the signaturefam argument. Both family arguments
     must have the same number of members. The signature of a
     family is described in a later section called "Family
     Signatures". FamCast is loosely named after the cast
     operator in the C programming language.

WriteFile(family, filename)
WriteFile(family, filename, "description")
     Writes a family file in the \ft\userdef directory. The
     input can be a scalar family or a vector family.

Save(family)

Saves a family variable so the family and its values can be used in an ExprFamilyTrade block. Normally the variables in an Expression block "disappear" at the end of the block. Save allows a vector family to persist and be visible for a subsequent ExprFamilyTrade block. The ExprFamilyTrade block does not have to be in the same script file as the Save() command, but it does have to be read during the same invocation of the Trade program. There is no limit on the number of families that can be Save'd.

Delete(name, ...)

Delete a name or names from Trade's symbol table and frees up the memory they had consumed. This is most useful for deleting large family variables after they are not needed. Delete does not delete files from the disk. It only deletes Trade variables. If you are not working with large families or your PC has plenty of RAM then you won't need to use this.

It is not necessary to use Delete() at the end of an [Expression] block. Trade deletes all symbols and frees the memory they had used at the end of a block.

FamAvg(family)

If the input is a vector family, FamAvg returns a vector with a daily return that is the average of the daily returns of all of the vectors in the family. If the input is a scalar family, FamAvg returns a scalar that is the average value of the input scalars.

FamAD(family)

Returns a vector which is the difference on each day between the number of advancing funds in the family minus the number of declining funds in the family.

Family Examples

The next example builds a family file of no-load equity funds available from Fidelity.

```
[Expression]

;Select only equity funds from B-FI-RET.
fam = b-fi-ret FamAnd all-eq

;Exclude load funds.
fam = fam FamAndNot load-avg
fam = fam FamAndNot load-hi
fam = fam FamAndNot load-low
fam = fam FamAndNot load-na

;Write the family file
WriteFile(fam, fidnleq, "Fidelity no-load equity funds")
```

The next example prints a table of a family's percent rate of change for one day, one week, one month, and one year. The table is sorted on the one day column.

```
[Expression]
fam = select    ;Fidelity select family

roc_1d = Roc(fam, 1) * 100
roc_1w = Roc(fam, 5) * 100
roc_1m = Roc(fam, 22) * 100
roc_1y = Roc(fam, 254) * 100

FamPrint(Sort(roc_1d), "1D %Chg",
        roc_1w, "1W %Chg",
        roc_1m, "1M %Chg",
        roc_1y, "1Y %Chg")
```

and the first few lines of output on 6/2/98 are

		1D %Chg	1W %Chg	1M %Chg	1Y %Chg
FSPFX	Fidelity Sel Paper & Forest/50	2.04	-3.84	-5.38	7.39
FDCPX	Fidelity Sel Computers/007	1.91	-2.80	-7.75	12.68
FSCSX	Fidelity Sel Software-Comp/028	1.70	-0.49	-8.68	20.34
FSAGX	Fidelity Gold Portfolio/041	1.14	-6.44	-16.73	-36.29
FSPTX	Fidelity Sel Technology/064	1.02	-2.13	-8.92	8.56
FSDAX	Fidelity Sel Defense & Aero/06	1.01	0.14	-7.04	27.88
FSDCX	Fidelity Sel Develop Commun/51	1.01	-0.59	-5.51	16.82
FSELX	Fidelity Sel Electronics/008	1.01	-4.26	-15.34	-5.66
FSAIX	Fidelity Sel Air Transpor/034	0.66	-1.30	-4.83	38.38

The next example combines the VALUE and CAP-SMAL families, then creates a family subset of only those funds with a 2 year ulcer index less than 3.5 and 1.5 year Ann greater than 20. Configurable parameters such as the family names, and the number of years to use for UI and Ann are assigned at the beginning so they can be easily changed.

```
[Expression]
fam = value FamOr cap-smal
UIyears = 2
UImax = 3.5
Annyears = 1.5
```

```
AnnMin = 20

;Compute ulcer index
DaysPerYear = 253
peak = Max(fam, UIyears * DaysPerYear)
cdd = (peak - fam) / peak
ui = 100 * Pow(Sma(cdd * cdd, UIyears * DaysPerYear), .5)

;Remove funds with ulcer index > UImax
lowUI = Select(ui < UImax)

;Compute Annual return
Ann = Last(fam) / Last(Shift(fam, Annyears * DaysPerYear))
Ann = 100 * (Pow(10, Log(Ann)/ Annyears) - 1)

;Remove low Ann funds from the lowUI family.
Ann = Select(Ann > AnnMin) FamAnd lowUI

;Print the family Ann and UI, sorted by Ann
FamPrint(Sort(Ann), "Ann", lowUI, "UI")
```

The script produced the following output on 8/17/98.

		Ann	UI
OAKVX	Oak Value	26.43	2.41
FAMVX	Fam Value	23.52	3.20
LLSCX	Longleaf SmallCap	22.55	2.00
SLVAX	Seligman LargeCap-A	22.39	2.98
PNVAX	Putnam Intern'l Voyager-A	22.17	3.32
MCOFX	MFS Capital Opportunity-A	22.06	2.95
DFCSX	Dfa Continental Small Co	21.69	2.81
AVLFX	Aim Value-A	21.66	3.46
PLCVX	PBHG LargeCap Value	21.49	3.02
VUSVX	Vontobel US Value	21.05	1.94
RLVAX	Reserve LargeCap Value Equity	20.63	2.64

The next example removes high ulcer index funds from the GROWTH family, then computes a weighted score for each fund using four indicators -- Accutrack, a stochastic histogram, Rsi, and Volatility. These four indicators are computed for the family. Then scaled versions of the indicators a computed such that the best fund has a value of 1.0 and the worst fund has a value of 0. The four scaled indicators are multiplied by their weights and summed, and this sum is divided by the sum of the weights and multiplied by 100 so that the range of possible scores is between 0 and 100. The score along with each of the components are printed in a table, sorted by the score value. This script can be used as a template to develop a weighted score of any indicators.

```
[Expression]
fam = growth
AccuWeight = 9
```

```
VolWeight = 6
StocWeight = 5
RsiWeight = 4
UImax = 4
UIdays = 254
VolDays = 500

;Compute the ulcer index for each fund.
peak = Max(fam, UIdays)
cdd = (peak - fam) / peak
ui = 100 * Pow(Sma(cdd * cdd, UIdays), .5)

;Remove high UI funds from the family.
fam = fam FamAnd Select(ui < UImax)

;Compute volatility of each fund
volatility = 100 * Stdev(Roc(fam, 1), VolDays)

;Compute stochastic histogram for each fund
Average = 53
Smooth = 49
Trigger = 28
stoch = 100 * (fam - Min(fam, Average)) /
      (Max(fam, Average) - Min(fam, Average))
stoch = Ema(stoch, Smooth)
average = Ema(stoch, Trigger)
StocHisto = stoch - average

;Compute Rsi22 for each fund
Rsi22 = Rsi(fam, 22)

;Compute AT10, 53 for each fund against a horizontal index
accu1053 = Accu(fam, Irate(0), 10, 53)

;Get max and min values for the 4 indicators.
AccuMax = Max(Last(accu1053))
AccuMin = Min(Last(accu1053))
StocMax = Max(Last(StocHisto))
StocMin = Min(Last(StocHisto))
RsiMax = Max(Last(Rsi22))
RsiMin = Min(Last(Rsi22))
VolMax = Max(Last(-volatility))
VolMin = Min(Last(-volatility))

;Compute an overall family score based on the weights defined at top.
AccuScaled = (Last(accu1053) - AccuMin) / (AccuMax - AccuMin)
StocScaled = (Last(StocHisto) - StocMin) / (StocMax - StocMin)
RsiScaled = (Last(Rsi22) - RsiMin) / (RsiMax - RsiMin)
VolScaled = (Last(-volatility) - VolMin) / (VolMax - VolMin)

;Compute a score for each fund.
FamScore = (AccuScaled * AccuWeight +
            StocScaled * StocWeight +
            RsiScaled * RsiWeight +
```

```
                VolScaled * VolWeight) * 100 /
                  (AccuWeight + StocWeight + RsiWeight + VolWeight)

    ;Print the family score and its components.
    FamPrint(Sort(FamScore), "Score",
        accu1053, "AT10/53",
            StocHisto, "StocHisto",
            Rsi22, "RSI 22",
            volatility, "Volatlty")
```

The script produced the following output on 7/17/98.

	Score	AT10/53	StocHist	RSI 22	Volatlty
FCNTX Fidelity Contrafund/022	82.73	66.74	8.11	76.31	0.81
AVLFX Aim Value-A	82.09	72.65	6.60	77.00	0.88
AVLBX Aim Value-B	82.02	71.86	6.89	76.85	0.88
JAENX Janus Enterprise	81.54	68.35	9.82	73.83	0.89
RBCAX Reserve Blue Chip Growt	81.45	71.27	10.67	74.68	0.97
STRFX Strong Total Return	80.71	74.20	6.95	75.52	0.94
CMSTX Columbia Common Stock	80.31	62.30	8.93	72.87	0.83
CLMBX Columbia Growth	80.28	72.64	8.27	74.38	0.96
SGRWX UAM Instl:Sirach Growth	80.01	68.00	7.48	74.31	0.89
MCGAX MFS LargeCap Growth-A	79.52	62.54	8.50	73.73	0.86

The next example shows an application of the FamSum function. It creates a signal using the accutrack rank of a money market fund within a family. A sell signal is generated when fdrxx is ranked in the top MMpcnt percent of a family (in this example the select family is used).

```
    [Expression]
    fam = select.fam

    ;Define sell when MM rank is in top 20% of family
    MMpcnt = 20
    Fastfilt = 12
    Slowfilt = 48

    fam = fam FamAndNot Family(fdrxx) ;Remove MM from family
    Famaccu = Accu(fam, fdrxx, Fastfilt, Slowfilt)
```

```
numFunds = FamSum(fam - fam + 1)
mmRank = 1 + FamSum(0.5 + Sgn(Famaccu))
mmsig = Signal((mmRank - 1) / numFunds - MMpcnt / 100)
WriteFile(mmsig, mmsig)
```

Performance

A Trade vector uses approximately 10K bytes of RAM, so expressions of large families require sufficient RAM for efficient operation. For example, a vector family of 500 members uses at least 5M bytes, and a typical Expression block needs space for local variables and intermediate expression results. As a general rule of thumb, 32M bytes of RAM is needed for expressions of medium complexity using families of around 300 members. If sufficient RAM is not available, the PC uses disk swap space for virtual memory, and your script will run very slowly. If your computer has 32M byte or less RAM and your family scripts are slow because of disk swapping, buy an additional 64M byte of RAM and you wont have to be concerned about RAM space. With the price of RAM currently down to about $1 per megabyte, there is no reason to have to suffer with insufficient RAM.

There are some things you can do to conserve RAM if you are working with large families.

- Quite often you will only care about the values that a family has on the most recent FT day. This means you can use a scalar family instead of a vector family in your calculations. A scalar family uses very little RAM. Use the Last() function to make a scalar family from a vector family. Operations on scalar families are also much faster than on vector families.

- Use Select() and the family set operators (FamAnd, FamOr, FamAndNot) to reduce the size of your family variables as early as possible.

- Don't make many intermediate variables of large vector families. Each variable chews up RAM. Organize your expressions on large families in as few lines as possible. For example, if foo is a family, then

  ```
  yday = Shift(foo, 1)
  diff = foo - yday
  ratio = (yday - diff) / yday
  pcnt = ratio * 100
  ```

  needs memory for 7 copies of foo -- one each for foo, yday, diff, ratio, and pcnt, plus two more temporary copies for intermediate calculations that are freed up after each operation. Re-writing the expression as

```
pcnt = 100 * (foo - Shift(foo, 1)) / foo
```

consumes only 4 times the size of foo, and half of that is freed up after the operation.

- Use the Delete() function to delete large family variables after they are no longer needed. This frees up their RAM for other uses. Note that there is no reason to do this at the end of an Expression block -- Trade automatically does this.

The example below shows some of these techniques for conserving RAM. It is one of the previous family examples, modified for use with the large family B-FI-RET. Each large temporary family variable is Deleted as soon as it is no longer needed, and the family of final interest is made successively smaller as funds with high ulcer index and then funds with low annual return are excluded.

```
[Expression]

fam = b-fi-ret
Delete(b-fi-ret)    ;delete large unneeded family variables
UIyears = 2
UImax = 3.5
Annyears = 1.5
AnnMin = 20

;Compute ulcer index
DaysPerYear = 253
peak = Max(fam, UIyears * DaysPerYear)
cdd = (peak - fam) / peak
Delete(peak)    ;delete large unneeded family variables
ui = 100 * Pow(Sma(cdd * cdd, UIyears * DaysPerYear), .5)
Delete(cdd)    ;delete large unneeded family variables

;Remove funds with ulcer index > UImax
lowUI = Select(ui < UImax)
Delete(ui)    ;delete large unneeded family variables

;Make a copy of the lowUI fund but with price for values
Ann = fam FamAnd lowUI
Delete(fam)    ;delete large unneeded family variables
Ann = Last(Ann) / Last(Shift(Ann, Annyears * DaysPerYear))
Ann = 100 * (Pow(10, Log(Ann) / Annyears) - 1)

;Remove low Ann funds from the lowUI family.
Ann = Select(Ann > AnnMin) FamAnd lowUI

;Print the family Ann and UI, sorted by Ann
FamPrint(Sort(Ann), "Ann", lowUI, "UI")
```

Family Signatures

Note that in an earlier example, one of the lines of the ulcer index expression contained several vector families. In the expression

```
cdd = (peak - fam) / peak
```

the variables peak and fam are both family vectors. As a general rule, you can have multiple families within an expression as long as the families all have the same funds and in the same sequence. The combination of the list of funds within a family and their order is called a "signature". Families with the same signature can be used together within an expression. Families with different signatures cannot.

If the example above is modified as in

```
cdd = (peak - fam) / Sort(peak)
```

then an error is generated, since Sort(peak) is a family with a different signature than peak and fam.

Most Trade operators and functions that operate on families return a family with the same signature as their inputs. This includes all arithmetic and boolean logic functions. Only the following can return a family with a different signature than its input family(s):

```
Select()
Sort()
FamAnd
FamCast()
FamOr
FamAndNot
```

The FamAnd operator is unique in this list, because it returns a family that has the same signature as its second argument as long as the second family argument is a subset of the first argument. This property can be very useful if you need to make a copy of a family that has the same signature but with different values. In other words, if fam2 is a subset of fam1 (all the symbols in fam2 are contained in fam1), then after

```
newfam = fam1 FamAnd fam2
```

newfam has the same signature as fam2 but has the values taken from fam1.

Undefined Values

When working with the -d option to test Expression scripts at earlier dates, you will often have funds in a family that were not in existance at the test date. If a fund does not have any data

at the date Trade is given to be the most recent, the data for the fund is considered undefined by Trade, and the data for any variables derived from an undefined fund is also considered to be undefined. The values of an undefined fund are printed using dashes. For example, the following script

```
[Expression]
Print(oakmark, "$sym")
```

when run with the command line option "-d 1/1/95" produces the output

	12/30/94	12/29/94	12/28/94	12/27/94	12/23/94
OAKMX	14.14	14.05	13.99	13.97	13.96
OAKIX	7.91	7.87	7.85	7.85	7.96
OAKSX	----	----	----	----	----
OAKEX	----	----	----	----	----
OAKBX	----	----	----	----	----
OAKLX	----	----	----	----	----

Writing an undefined vector to an fnu file will produce a valid file with a header but no data fields. Undefined vectors are not considered an error condition and will not cause Trade scripts to halt prematurely.

When a scalar is derived from an undefined vector, the value of that scalar is also undefined, and the results from any expressions involving the scalar are undefined. Trade uses a special token internally to mark the undefined scalar so it will not try use its value and produce an incorrect numeric result. As with the undefined vectors, the value of an undefined scalar is printed as dashes, and undefined scalars will not halt Trade processing.

Family functions such as FamAvg, FamSum, and FamAD that perform cumulative operations on all family members generally ignore family members that are undefined on the date being processed.

Undefined family members will always appear at the end of a family after a Sort operation.

Undefined family members will never be returned by the Select function. You can use this property to remove undefined family members with something like

```
fam = Select(fam <= 1e30)
```

Syntax Summary

This section contains a summary of all the argument types and return types of Expression functions and operators. This is simply reference material, and you shouldn't need it to write

Trade scripts.

Notation:

sc	Scalar
v	Vector
sig	Signal
str	Quoted String
date	Date
scf	Scalar Family
vf	Vector Family
sigf	Signal Family
|	Or (eg. v | sc means vector or scalar type)

For example, the entry

```
vf = Accu(v, vf, sc, sc)
```

means that the function Accu returns a Vector Family when the input arguments are a Vector, Vector Family, Scalar, and Scalar.

```
v = Accu(v, v, sc, sc)
vf = Accu(vf, v, sc, sc)
vf = Accu(v, vf, sc, sc)
vf = Accu(vf, vf, sc, sc)

v = v And v
vf = v And vf
vf = vf And v
vf = vf And vf
sig = sig And sig
sigf = sig And sigf
sigf = sigf And sig
sigf = sigf And sigf

v = Days(sig)
vf = Days(sigf)

Delete(sc | scf | v | vf | sig | sigf, ...)

sig = Edit(sig, day | date, Buy | Sell)
sig = Edit(sig, day | date .. day | date, Buy | Sell)
sigf = Edit(sigf, day | date, Buy | Sell)
sigf = Edit(sigf, day | date .. day | date, Buy | Sell)
v = Edit(v, day | date, sc)
v = Edit(v, day | date .. day | date, sc)
vf = Edit(vf, day | date, sc)
vf = Edit(vf, day | date .. day | date, sc)

v = Ema(v, sc)
v = Ema(v, v)
vf = Ema(v, scf)
vf = Ema(v, vf)
vf = Ema(vf, sc)
vf = Ema(vf, scf)
```

```
vf = Ema(vf, v)
vf = Ema(vf, vf)

v = Emaz(v, sc)
v = Emaz(v, v)
vf = Emaz(v, scf)
vf = Emaz(v, vf)
vf = Emaz(vf, sc)
vf = Emaz(vf, scf)
vf = Emaz(vf, v)
vf = Emaz(vf, vf)

v = FamAD(vf)

scf = scf FamAnd scf
scf = scf FamAnd vf
vf = vf FamAnd scf
vf = vf FamAnd vf

scf = scf FamAndNot scf
scf = scf FamAndNot vf
vf = vf FamAndNot scf
vf = vf FamAndNot vf

v = FamAvg(vf)

scf = FamCast(scf, scf | vf | sigf)
vf = FamCast(vf, scf | vf | sigf)
sigf = FamCast(sigf, scf | vf | sigf)

scf = scf FamOr scf
vf = vf FamOr vf

scf = FamPos(scf)
scf = FamPos(vf)
scf = FamPos(sigf)

FamPrint(vf | scf | sigf, str, ...)

v = FamSum(vf)
sc = FamSum(scf)

scf = Family(sc, ...)
vf = Family(v, ...)

sc = First(v)
sc = First(scf)
scf = First(vf)

v = GetMember(vf, v)
sc = GetMember(scf, sc)
sig = GetMember(sigf, sig)

If (sc)
```

```
If (v)
If (sig)

v = Irate(sc)
vf = Itrate(scf)

sc = Last(v)
sc = Last(scf)
scf = Last(vf)

sc = Log(sc)
v = Log(v)
scf = Log(scf)
vf = Log(vf)

v = Lreg(v, sc)
vf = Lreg(v, scf)
vf = Lreg(vf, sc)
vf = Lreg(vf, scf)

sc = Max(scf)
v = Max(v, sc)
v = Max(v, v)
vf = Max(v, scf)
vf = Max(vf, sc)
vf = Max(vf, scf)

sc = Min(scf)
v = Min(v, sc)
v = Min(v, v)
vf = Min(v, scf)
vf = Min(vf, sc)
vf = Min(vf, scf)

sig = Not sig
sigf = Not sigf
v = Not v
vf = Not vf

v = v Or v
vf = v Or vf
vf = vf Or v
vf = vf Or vf
sig = sig Or sig
sigf = sig Or sigf
sigf = sigf Or sig
sigf = sigf Or sigf

sc = Pow(sc, sc)
v = Pow(sc, v)
v = Pow(v, sc)
v = Pow(v, v)
scf = Pow(sc, scf)
scf = Pow(scf, sc)
```

```
scf = Pow(scf, scf)
vf = Pow(sc, vf)
vf = Pow(v, scf)
vf = Pow(v, vf)
vf = Pow(vf, sc)
vf = Pow(scf, v)
vf = Pow(vf, v)
vf = Pow(vf, vf)

Print(str, ...)
Print(sc | v | scf | vf | sig | sigf, str, ...)

v = Roc(v, sc)
vf = Roc(v, scf)
vf = Roc(vf, sc)
vf = Roc(vf, scf)

v = Rsi(v, sc)
vf = Rsi(v, scf)
vf = Rsi(vf, sc)
vf = Rsi(vf, scf)

Save(vf)

scf = Select(scf test sc)       test is one of <, <=, >, >=
vf = Select(vf test sc)

v = Sgn(v)
vf = Sgn(vf)
sc = Sgn(sc)
scf = Sgn(scf)

v = Shift(v, sc)
vf = Shift(v, scf)
vf = Shift(vf, sc)
vf = Shift(vf, scf)

sig = Signal(v)
sigf = Signal(vf)

v = SignalPairTrade(v, v, sig)
vf = SignalPairTrade(vf, v | vf, sig | sigf)
vf = SignalPairTrade(v | vf, vf, sig | sigf)
vf = SignalPairTrade(v | vf, v | vf, sigf)
v = SignalPairTrade(v, v, sig, sc)
vf = SignalPairTrade(vf, v | vf, sig | sigf, sc)
vf = SignalPairTrade(v | vf, vf, sig | sigf, sc)
vf = SignalPairTrade(v | vf, v | vf, sigf, sc)

v = Sma(v, sc)
vf = Sma(v, scf)
vf = Sma(vf, sc)
vf = Sma(vf, scf)
```

```
scf = Sort(scf)
vf = Sort(vf)
sigf = Sort(sigf)

v = Stdev(v, sc)
vf = Stdev(v, scf)
vf = Stdev(vf, sc)
vf = Stdev(vf, scf)

sig = Stretch(sig, sc)
sigf = Stretch(sigf, sc)

v = Sum(v)
vf = Sum(vf)

v = Vector(sig)
vf = Vector(sigf)

v = Warp(v, sc, sc)
vf = Warp(vf, sc, sc)
vf = Warp(v, scf, sc)
vf = Warp(vf, scf, sc)

WriteFile(v | sig | scf | vf | sigf, name)
WriteFile(v | sig | scf | vf | sigf, name, str)

sig = ZigZag(v, sc)
sigf = ZigZag(v, scf)
sigf = ZigZag(vf, sc)
sigf = ZigZag(vf, scf)

sc = sc op sc              op is one of +, -, *, /
v = sc op v
scf = sc op scf
vf = sc op vf
v = v op sc
v = v op v
vc = v op scf
vc = v op vf
scf = scf op sc
vf = scf op v
scf = scf op scf
vf = scf op vf
vf = vf op sc
vf = vf op v
vf = vf op scf
vf = vf op vf
```

[T76], [T17], [T87]

These blocks create the Dennis Meyers T76, T17, and T87 timing signal files and also display a recent history of the signal indicators. See the files T76.DOC and T17.DOC for a detailed

description of these two signals.

Note that you can also use the [Expression] commands to create T76, T17, T87, and other complex signals using their basic component indicators. The files T76EXPR.INI, T17EXPR.INI and T87EXPR.INI are examples.

[Portfolio]

This command allows you to create FNU files from your portfolio trading history, allowing you to view your portfolio growth as a FastTrack plot, and compare it with other indices and funds. It models both cash and margin type accounts. Two FNU files are created. One is simply the daily value of the portfolio. The other is the daily gain of the portfolio independent of amounts deposited or withdrawn from the portfolio account.

See the file PORTF.DOC for a complete description of this command block.

[AccuRank]

This block displays a sorted ranking of a family using accutrack, using a format which is very similar to Brian Stock's excellent FastTools program. The displayed accutrack values are scaled by the value 26500, making them identical to those in FastTools. Optional commands allow you to display rank position instead of accutrack value, change the interval with which previous days values are displayed, and change the most recent day in the table to something other than the most recent day of the FastTrack database.

Use the command

```
trade -h4 >rank.ini
```

to create a template of an [AccuRank] command file, then edit the file. Commands which are commented out in this template are optional.

[SlopeRank]

This block displays a sorted ranking of a family using one of 3 different slope indicators. It has the same format as the AccuRank block. The 3 slope indicators are

DeltaPosition	This is the slope indicator that is used by Roy Ashworth's FastRube. It is based on the change of a fund's position in an accutrack sort between 2 dates. In FastTools, this is ranking based on the K (chg) column of the E

module, Rel Strength Position.

DeltaValue — This indicator is based on a fund's change in accutrack value between 2 dates. In FastTools, this is ranking based on the K (chg) column of the F module, Rel Strength Values.

DeltaValueEma — This indicator is formed by taking the single day change of a fund's accutrack value and smoothing it with an EMA. The values are multiplied by

```
(SlopeFilter + 1) / 2
```

This has no affect on the relative rankings, but it scales the indicator values so that typically they have the same order of magnitude as accutrack. This also means that if you use very large values of SlopeFilter, the DeltaValueEma indicator becomes the same as accutrack.

Slope indicators are interesting because they give earlier buy signals than accutrack. One of the reasons I wrote Trade was the belief that slope indicators could be used to advantage in an automated trading system, and also that the particular slope indicator used in FastRube was not a very good one. Subsequent testing has proven this out. The other two slope indicators consistently give better returns by several percentage points in backtesting. The DeltaValueEma gives the best returns of the three, and is the easiest to control.

Use the command

```
trade -h5 >rank.ini
```

to create a template of an [SlopeRank] command file, then edit the file. Commands which are commented out in this template are optional.

[AccuFamilyTrade]

This command block controls a Family trading strategy based on Accutrack ranking. Its operation is similar to the FastRube program without moving averages and parabolics, but with the ability to automate running a strategy a number of times by iterating one or more parameters over a range of values.

The operation of this block involves holding a set number of positions which are traded according to your specified buy and sell criteria. The choice of funds is limited to those in the your specified Family, plus your money market fund. The money

market fund does not have to be one of the funds in the family. The buy and sell criteria can optionally include a signal file. On each day, the following trading algorithm is followed.

- The funds in the family and the money market fund are ranked based on their Accutrack values. The fund with the highest accutrack value gets a rank of 1, next highest 2, etc.

- Each position is first checked for sell conditions. Any positions that have met the sell criteria are sold and replace with the money market fund.

- Each position is then checked for buy conditions. Any position that is holding the money market fund will buy the best ranked fund (lowest number rank) that meets the buy criteria.

The sell criteria is defined as:

```
(Signal is Sell) OR ((days held >= HoldMinDays) AND
       ((Rank > HoldMaxRank) OR (Rank > money market rank)))
```

The buy criteria is defined as:

```
(Signal is Buy) AND (Rank <= BuyMinRank) AND
                        (Rank < money market rank)
```

The buy and sell rank criteria are set by the BuyMinRank and HoldMaxRank commands. There are two ways to specify these criteria. If you set them without using a percent sign, the number represents an absolute rank. For example, if you give

```
HoldMaxRank = 14
```

then a fund will be sold when its rank in the family becomes worse than 14 (15 or greater). This can be a bit constraining when a family has funds that did not exist over the life of the FastTrack database. For example, if a family has 30 funds but only 18 of them existed on 9/1/1988, then a HoldMaxRank of 14 has quite a different effect in 1988 than it does now. You can give the ranks as a percentage of the number of funds in the family by adding a trailing percent sign (%) to the rank number. In this example

```
HoldMaxRank = 40%
```

means that a fund will be held until its rank becomes worse than 40% of the funds that were in the family on each day.

If you set up identical strategies in Trade and FastRube using the same parameter values including the StartDate you will get the same sequence of buys and sells. You must also check the "Initial Buy Delayed" option under FastRube's "Portfolio

Calculation Options". You may see a small difference in Total value at the end of the run because of a difference in the way FastRube and trade treat money market funds. By default, Trade credits interest accrued in money market funds daily, while FastRube credits it monthly. If you give the optional

```
RubeCompatibility = 1
```

command, it sets the default start date to the same as FastRube's, and it credits money market interest monthly instead of daily. In the output display of buys and sells, Trade normally lists all trades including the MMarket trades. FastRube does not list the MMarket trades. If you set the RubeCompatibility option to 1, Trade excludes the MMarket trades from the output list.

Trade also separates the PenaltyDays from HoldMinDays, so results accurately reflect short term penalties with Fidelity Select funds if you set the HoldMinDays to less than 30.

The easiest way to create an [AccuFamilyTrade] command block is to redirect the output of the online help screen to a file, then edit the file. For example, you can create a template for a command file using

```
trade -h6 >accu.ini
```

The commands that are commented out with a leading semicolon in the template file are optional. After editing the parameters in accu.ini, you can run it with standard output redirected to a list file using

```
trade accu.ini >lst
```

When the run is done, the trade results can be viewed in the text file named lst. You can also create an FNU file of the trading history and view it in FastTrack.

[SlopeFamilyTrade]

This command block controls a Family trading strategy based on one of the 3 slope indicators described above under [SlopeRank]. Except for the different ranking indicator used and differences in the buy and sell criteria, its operation is similar to the AccuFamilyTrade block.

In the SlopeFamilyTrade block, the rank used for the buy criteria is one of the 3 slope indicators. The rank used for the sell depends on whether RelativeRankSell is set to 1 or 0. An additional buy criteria is also added when RelativeRankSell is set to 1.

The rank of the money market fund is ignored in SlopeFamilyTrade

operation.

The latest version of FastRube has added Slope Options which are equivalent to command options in Trade. The corresponding options are

Trade Command	FastRube Slope Option
SlopeRankType = DeltaPosition	Position
SlopeRankType = DeltaValue	Value
SlopeRankType = DeltaValueEMA	Value Ema
RelativeRankSell = 0	Sell on Slope selected
RelativeRankSell = 1	Sell on Slope un-selected

RelativeRankSell = 1

If RelativeRankSell is set to 1, the rank criteria for sell is based on difference between a position's accutrack rank and the best accutrack rank the position had attained. Recall that all of the slope indicators are based on a change in an accutrack indicator. When a fund is bought, its best accutrack numeric rank (lowest rank number) is tracked. A position meets the rank part of the sell criteria when the difference between its accutrack rank and its best accutrack rank exceeds the HoldMaxRank setting.

For example, a position is bought and the HoldMaxRank is set to 10. During the period of owernership, the accutrack ranking rises to a best value of 3. The fund will meet the rank part of the sell criteria when its accutrack ranking falls below (numerically higher than) 13.

The sell criteria is defined as:

```
(Signal is Sell) OR ((days held >= HoldMinDays) AND
           (Accu Rank fall from Max > HoldMaxRank))
```

The buy criteria is defined as:

```
(Signal is Buy) AND (Slope Rank <= BuyMinRank) AND
           (Accu Rank <= FamilySize - HoldMaxRank)
```

RelativeRankSell = 0

If RelativeRankSell is set to 0, the rank indicator for sell is the same as the rank indicator for buy. This is the selected SlopeRankType.

The sell criteria is defined as:

```
(Signal is Sell) OR ((days held >= HoldMinDays) AND
           (Slope Rank > HoldMaxRank))
```

The buy criteria is defined as:

```
(Signal is Buy) AND (Slope Rank <= BuyMinRank)
```

[ExprFamilyTrade]

This command block controls a Family trading strategy based on a family of values that are computed in an Expression block. Its operation is very similar to the AccuFamilyTrade block, but you have the freedom to create any arbitrary indicator to use for the daily ranking. In an Expression block the family of vector values is saved using the Save() function, which allows it to be referenced in the ExprFamilyTrade block. The example below shows how you can create an automated trading strategy that buys the funds whose 11-day Emas are the highest percentage above their 39-day Emas.

```
[Expression]
fam = select
HolyGrail = 100 * (Ema(fam, 11) - Ema(fam, 39)) / Ema(fam, 39)
Save(HolyGrail)

[ExprFamilyTrade]
Family = HolyGrail
MMarket = FGRXX
Positions = 3
BuyMinRank = 4
HoldMaxRank = 40%
HoldMinDays = 30
```

The sell criteria is defined as:

```
(Signal is Sell) OR ((days held >= HoldMinDays) AND
                     (Rank > HoldMaxRank))
```

The buy criteria is defined as:

```
(Signal is Buy) AND (Rank <= BuyMinRank)
```

The ExprFamilyTrade block assumes that your indicator values are "higher is better". If your Saved()'d indicator is such that lower numbers are better, simply negate the family before you save it.

As with the other family trade blocks, you must specify a money market fund using the MMarket command. If the MMarket that you give is not in the Save'd family, it will not have indicator values that you computed in the Expression block. In this case the indicator values for the money market fund are set to large negative numbers so it will always have the poorest ranking in the family and will not influence the BuyMinRank and HoldMaxRank

buy and sell rules.

Fund/Index Pair Trading

Trade has several command blocks which control the trading between a fund and an index using various TA indicators. These include Accutrack, Rsi, Bollinger Bands, and Linear Regression. All of these command blocks use a common set of controlling commands, and only differ in the setting of the parameters for the TA indicator which controls the trading. In each case you can specify an over-riding signal file and Money Market fund, create an fnu file of the trading results, and create a signal file of the buy and sell dates that were given by the controlling TA indicator.

You can also set the buy and sell delays to 0 day or 1 day. If you don't specify a delay, a 1 day delay is assumed. This means that you get your FT data in the evening after the market has closed, it gives a switch from one position to another, and you execute the trade on the following day. If you use some other data source during the day and anticipate and act on a switch before the market closes, they you are trading with a delay of 0 days and you should set the Delay = 0.

All of the Pair Trade blocks have a few required commands, and some additional optional commands.

Required Pair Trade Commands

Fund = name
: Sets the name of the symbol that will be used in the Fund position. This is the fund that will be purchased when the controlling TA indicator indicates a Buy.

Index = name
: Sets the name of the symbol that will be used in the Index position. This is the fund that will be purchased when the controlling TA indicator indicates a Sell.

Optional Pair Trade Commands

Delay = n
: Sets the buy and sell delay to 0 or 1 day. Any non-zero number will be interpreted as 1 day. If not given, 1 day trading delay is assumed. If you make your trading decisions in the evenings after the market has closed, you are trading with a delay of 1 day. If you follow realtime data during the day and make your trading decisions before the market closes, you are trading with a delay of 0 days.

HoldMinDays = n
: Sets the minimum number of calendar days that the fund or index will be held before a switch will be made. The

HoldMinDays does not delay switching to or from a Money Market fund if an optional signal file is used. If not given, no minimum hold period is enforced.

FnuFile = file
Sets the name of an optional fnu file which tracks the growth of an original $1000 investment at the start of trading. If not given, an fnu file is not created.

CreateSignal = file
Sets the name of an optional signal file which has the buy and sell dates indicated by the controlling TA indicator. These dates are the actual dates that the indicator switched and are not affected by Delay, HoldMinDays, or the use of a Signal file. If not given, a signal file is not created.

PenaltyDays = n
Sets the minimum number of calendar days that the fund or index must be held in order to avoid a short term trading penalty. If the fund or index is held less than PenaltyDays, the position is charged a short term penalty that is given in percent by the Penalty command. If not given, no short term penalties are imposed.

Penalty = n
Sets the percent penalty that a position is charged if it is sold after being held for less than PenaltyDays.

HistoryDays = n
By default, Trade displays all the trades that occurred since the StartDate. The HistoryDays command limits the display of trading history to the last n market days. If HistoryDays is set to 0, then all settings, history, and trading summary displays are suppressed.

StartDate = date
By default, Trade starts the trading on the first date that both the Fund and Index were available. The StartDate command allows the start of trading to begin any time after the Fund and Index were available.

StopDate = date
By default, Trade runs the Pair Trade strategies up to the most recent day in the FT database. The StopDate command allows the trading to be stopped before this date. Note that this can also be accomplished using the -d option on the Trade command line.

Signal = name
Sets the name of a signal file which will control when the strategy switches to a Money Market fund. If a Signal is given, the MMarket command must also be given.

MMarket = name

Sets the name of a money market fund which the Pair Trade strategy will switch to when the Signal indicates a Sell.

[AccuPairTrade]

This command block uses Accutrack as the controlling TA indicator in a pair trading scheme. The Fund is bought when Accutrack is positive, and the Index is bought when Accutrack is negative. (Trade uses an Accutrack calculation that is centered around the value 0, not 50 as in FT charts). The Accutrack filter constants are set by the command

AccuFilter = f1, f2

where f1 and f2 are the number of days in the two Accutrack Ema filters.

[SignalPairTrade]

This command controls the switching between a fund and an index using a signal file. The primary purpose for this is to create an FNU file of the trading results. A Signal command is required, and the following optional Pair Trade commands can be used: Delay, HoldMinDays, PenaltyDays, Penalty, StartDate, StopDate, FnuFile.

[BBandTrade]

This command block uses Bollinger Bands as the controlling TA indicator in a pair trading scheme. Buy and sell thresholds are set as offsets in standard deviations from the middle band. The middle Bollinger Band is simply the Ema of the daily prices. The time constant of this Ema is given by the command

FilterDays = n

Standard deviation of price is computed daily for the last n days given by the command

LookbackDays = n

The buy and sell threshold is given by the BuySellLevel command, which can be used with one or 2 arguments.

BuySellLevel = n

A buy is given when the price crosses n standard deviations below the middle Bollinger Band in the positive direction, and a sell is given when the price crosses n standard deviations above the middle Bollinger Band in the negative direction.

Two thresholds can be given, using the optional command form

BuySellLevel = n1, n2

When two thresholds are used, a buy is given when the price crosses n1 standard deviations below the middle Bollinger Band in the positive direction or if price drops n2 standard deviations below the middle band. A sell is given when the price crosses n1 standard deviations above the middle Bollinger Band in the negative direction or if price rises n2 standard deviaitons above the middle band.

For example, the command

    BuySellLevel = 1.1, 1.8

means a buy will be triggered when the Fund's NAV crosses 1.1 standard deviations below the middle Bollinger Band in a positive direction, or if it drops 1.8 standard deviations below the middle band.

[RsiTrade]

This command block uses Rsi as the controlling TA indicator in a pair trading scheme. Buy and sell thresholds are set as offsets from the middle Rsi value of 50. The number of days used for the filters in the Rsi calculation is given by the command

LookbackDays = n

The buy and sell threshold are given by the BuySellLevel command, which can be used with one or 2 arguments.

BuySellLevel = n

A buy is given when the Rsi crosses the value (50 - n) in the positive direction, and a sell is given when the Rsi crosses the value (50 + n) in the negative direction.

Two thresholds can be given, using the optional command form

BuySellLevel = n1, n2

When two thresholds are used, a buy is given when the Rsi crosses the value (50 - n1) in the positive direction or if it falls below (50 - n2), and a sell is given when the Rsi crosses the value (50 + n1) in the negative direction or if it rises above the value (50 + n2).

If in a RsiTrade block you give the command

    BuySellLevel = 5

a buy will be triggered when the Rsi value crosses 45 in a positive direction, and a sell and when it crosses 55 in a negative direction. In other words, a buy is not triggered until Rsi drops below 45, then rises above 45. Thus the value of 5 is the delta around the nominal value of 50 that sets both the buy and sell thresholds.

You can also give a second threshold. If you give the command

```
BuySellLevel = 4, 9.5
```

a buy will be triggered either when Rsi drops below 40.5, or when it crosses 46 in a positive direction, and a sell will be triggered either when Rsi rises above 59.5, or when it crosses 54 in a negative direction.

If you only want the type of buy sell triggers obtained with the second parameter, you can simply give a number larger than 50 for the first and it will never cause a buy or sell.

[LinearRegTrade]

This command block uses Linear Regression as the controlling TA indicator in a pair trading scheme. Buy and sell thresholds are set as percentage deviations from the best fit line through the last LookbackDays number of prices. The command

```
LookbackDays = n
```

sets the number of days that are used to compute the linear regression of price.

The buy and sell thresholds are given by the BuySellLevel command, which can be used with one or 2 arguments.

```
BuySellLevel = n
```

A buy is given when the price crosses the value of n percent below the linear regression line in the positive direction, and a sell is given when the price crosses the value of n percent above the linear regression line in the negative direction.

Two thresholds can be given, using the optional command form

```
BuySellLevel = n1, n2
```

When two thresholds are used, a buy is given when the price crosses the value of n1 percent below the linear regression line in the positive direction or if it falls n2 percent below the linear regression line, and a sell is given when the price crosses the value of n1 percent above the linear regression line in the negative direction or if it rises n2 percent above the

linear regression line.

For example, the command

```
BuySellLevel = 2, 5
```

means that a buy will be triggered if the price crosses the threshold that is 2% below the best fit line in a positive direction, or if the price drops more than 5% below the best fit line.

[AccuFamilyPeriod]

The purpose of this command is to create an FNU file which shows how strongly the market has trended as a function of time. The FNU it creates is the average time between a fund first meeting its accutrack buy criteria until its ranking falls and meets its sell criteria. When this time is long, the market is said to be "trending", and trend following trading strategies typically perform well. When this time becomes short, the market is said to be "choppy", and trend following strategies do not perform well.

To use this command, define the Family, AccuFilter time constants, BuyMinRank, and HoldMaxRank. It works particularly well with the Fidelity Selects family, but any family can be selected. For each day, the program looks at each fund in the family that had previously met the BuyMinRank criteria and has not yet fallen past the HoldMaxRank criteria. It averages the the number of days that have elapsed since they met the BuyMinRank criteria. At any given point in time, some of these funds will have just met the buy criteria, while others will be almost ready to meet their sell criteria, so the average hold time of these funds is one half what their total average hold times end up being. The computed average is therefore doubled for each day. A simple moving average is used to smooth the FNU plot. The constant for this SMA is set using the LookbackDays command.

Here's a simple exercise that I think you will find quite revealing. Create a command file using the parameters in the help screen using

```
trade -h11 > per.ini
```

Edit per.ini, setting the family to your own version of the Fidelity Selects minus the 2 metal funds (fsagx and fdpmx), then run the command with

```
trade per.ini >lst
```

This creates the FNU file BSPER.FNU. Run FastTrack, set the Fund to BSPER, and view the T chart from the beginning of the database

(PT, EA). What you will see is that the average hold times for a typical Accutrack family trading strategy have become very short in the past 2 years. To see how this has affected their performance, read the AccuFamilyEval command description below.

[AccuFamilyEval]

This command is similar to the AccuFamilyTrade command, except that instead of setting the number of positions to hold, it buys every fund that meets the buy criteria. While you wouldn't want to actually trade this way, it is my opinion that this is a much better way to evaluate the average performance of an Accutrack trading strategy than to look at individual runs of holding a small fixed number of positions.

One of the problems with evaluating a strategy based on holding a small fixed number of positions is that you only actually end up buying a small percentage of the funds that meet the buy criteria, and the funds that you buy are very dependent on the StartDate. Because the rankings change almost daily, changing the StartDate sets each run off on a completely different course of trades. A typical hold 1 position strategy will make about 55 trades, but there are about 430 occurrences of funds crossing the BuyMinRank requirement (these numbers for example AT6/24, BuyMinRank 3). A lucky run will hit 55 trades with a high percentage of profitable trades and might show an annual return greater than 40%. An unlucky run with the exact same parameters but a different start date might show an annual return in the teens. They are both from the same set of 430 possible trades. The set of possible trades is determined as soon as you set the AccuFilter, BuyMinRank, and HoldMaxRank.

A hold 3 position strategy will make 3 times as many trades as the hold 1, and its variability will be correspondingly less, but there will still be a big difference between a lucky run and an unlucky run. The more funds that you hold, the more the results will converge around the true average performance of the strategy.

There is a popular belief among FastRube users that hold-1-position strategies have much higher returns than those that hold more positions. This belief stems from the fact that you can always find, if you try hard enough, one extremely lucky hold-1-position run that is way above the mean. These lucky runs get documented and passed around among users, some of whom draw the incorrect conclusion that these results will continue into the future. A little testing with the AccuFamilyEval command on the Fidelity Select family shows that the difference between the average performance of a typical hold-1-position strategy and a hold-5-position strategy is around 2% in annual return.

The same mechanism that makes individual runs of a hold 1, 2, or 3 position trading strategy so variable with StartDate is also in

effect when you change other parameters by a small amount. Suppose for example that you have found a hold-1-position strategy using Accutrack 6,24 that had an annual return of 30%, and when you change the filter parameters to 6,25 the annual return increased to 38%. Is the average performance of 6,25 really 8% better than 6,24? There's no way to tell from that information, because these are just 2 runs that picked 55 trades at random from about 430 potential choices. What you will find if you evaluate all 430 of the trades that met the buy criteria is that small changes in Accutrack filter such as this example have very little affect on average performance.

The problems of wildly varying results gets even worse when you evaluate the performance over only one year. A hold 1 position strategy will make about 7 trades in one year, so the results from one slightly different run to the next vary all over the place. When you use the AccuFamilyEval command, you are averaging the results of perhaps 50 trades over 1 year, so the results are much more stable.

Here's another simple exercise that will reinforce the one described for the AccuFamilyPeriod commands. Create a command file using the parameters in the help screen using

```
trade -h12 > eval.ini
```

Edit eval.ini, setting the family to your own version of the Fidelity Selects minus the 2 metal funds (fsagx and fdpmx), and remove the semicolon which comments out the HoldMinDays command so that it gets set to 30. Then run the command with

```
trade eval.ini >lst
```

This creates the FNU file EVAL.FNU. Run FastTrack, set the Fund to EVAL and the Index to VFINX, and view one year of the T chart (PT, EO). Use the left arrow key to move the 1 year display further back in time, and see how far you have to go before EVAL starts outperforming the index fund. You will have to go back to about the same time in '95 where the average hold times seen in PER.FNU were much longer.

Iteration

Many of the commands for the various Trade command blocks allow you to Iterate a parameter over a range of values, running the command block for each value. For example, if you want to run an AccuFamilyTrade command for each integer value of AccuFilter from 6,24 to 6,40, use the command

```
AccuFilter = 6, Iterate(24..40)
```

You can optionally set a stepsize, as in the example

```
AccuFilter = 6, Iterate(24..40, 0.5)
```

which iterates a parameter in steps of 0.5. You can also iterate over an enumerated list of items. For example, you can run any of the Trade blocks using several different Signals with

```
Signal = Iterate(T76, T17, DJUSP, MIRAT)
```

You can iterate a date as in the example

```
StartDate = Iterate(1/1/96..6/1/96)
```

When you use the Iterate command a summary file named TRADE.SUM is written. This file gives the annual gain and max drawdown for each iteration of the command.

You can put multiple Iterate commands in a command block, in which case the program runs all possible combinations of all iterated parameters. For example, if you use the commands

```
BuyMinRank = Iterate(2..4)
AccuFilter = Iterate(1..7), Iterate(20..40)
HoldMaxRank = Iterate(14..19)
```

the program will run all possible combinations of these values. That is (3 x 7 x 21 x 6) runs, or a total of 2646 runs. At an average of 3 seconds per run, this will take a little over 2 hours on a reasonably fast pentium.

If you use the Iterate command with the optional percent notation with BuyMinRank or HoldMaxRank, place the percent sign after the closing parentheses as in

```
HoldMaxRank = Iterate(35..45)%
```

Here's a few things to consider if you are iterating a large number of runs.

- The file to which you redirect standard output can get very large for thousands of runs. You may need megabytes or even tens of megabytes of free disk space to hold it. If you have no interest in viewing the output of each run and are only interested in the summary file, you can redirect standard output to the special file NUL. Whenever you do this, Windows just throws the bits away. The summary file TRADE.SUM has only one line per run so it never gets very large.

- If you do want to look at some of the runs in the output file, you will need a text editor that can handle large files. Windows95 Wordpad seems to work well for this purpose.

- If you let standard output go to the screen and ignore the scrolling, the run will take much longer than redirecting to a file or to NUL.

- It is often helpful to sort the TRADE.SUM file on the first column, which is annual gain. This is easily done with the sort utility that comes with Windows. You redirect both standard input and standard output when you use sort, as in the example

```
sort <trade.sum >sorted.sum
```

The following list of commands can be used with Iterate:

```
AccuFilter
BuyMinRank
BuySellLevel
Combine
Delay
FilterDays
HoldMaxRank
HoldMinDays
LookbackDays
PenaltyDays
Positions
RelativeRankSell
Signal
SlopeDays
SlopeFilter
StartDate
```

Trade Command Line Options

While most of the Trade commands are given in text command files, there are a few command line options for functions which are more interactive in nature, or which you probably wouldn't want to run on a regular basis. Command line options must be given before filenames. These options are described below.

-d date
: Make Trade operate as though the most recent date in the FastTrack database is the given date.

-t
: Do an integrity check on the FastTrack database and families. Each family is visited, and the funds in the family are read and checked for consistency. Any symbols in a family for which neither a fund in the database nor an FNU file can be found are displayed at standard output.

-h
: This option displays a list of all the block commands and shows how to get individual help on each of them.

-o file1 file2
: This option computes the overlap between 2 FamilyTrade output files. Overlap is computed as the percentage of days that each strategy held identical funds. The idea

behind this is that trading based on Slope typically picks different funds than trading based on Accutrack, so that by combining two independent strategies you can achieve enough diversification to make it worthwhile to use both in parallel. Here is an example of how you might use this option.

```
trade accu.ini >accu.lst        (Run an AccuFamilyTrade cmd)
trade slope.ini >slope.lst      (Run a SlopeFamilyTrade cmd)
trade -o accu.lst slope.lst >diff.lst  (compute the overlap)
```

-s signal fund

Evaluate a signal file applied to a fund. This gives some details of switching between a fund and VMFXX using the signal. A one day buy and sell delay is assumed between the signal dates and the trade dates.

Trade Keywords

The list below contains all Trade keywords. You should avoid using names in command files that exactly match these keywords in both spelling and case. If you must use one of the words for a filename, family name, or variable name, be sure to use a different capitalization for at least one of the characters.

- Accu
- AccuFamilyEval
- AccuFamilyPeriod
- AccuFamilyTrade
- AccuFilter
- AccuPairTrade
- AccuRank
- AdjustedFnuFile
- And
- Avg
- BBandTrade
- Buy
- BuyMinRank
- BuySellLevel
- CashIn
- CashOut
- Combine
- CombineFamilyTrade
- CombineRank
- CreateSignal
- Days
- Delay
- Delete
- DeltaPosition
- DeltaValue
- DeltaValueEMA
- DisplayRankPositions
- DumpDate

Edit
Else
Ema
Emaz
EndIf
Expression
ExprFamilyTrade
FamAD
FamAnd
FamAndNot
FamAvg
FamCast
FamOr
FamPos
FamPrint
FamSum
Family
FilterDays
First
FnuFile
Fund
GetMember
HistoryDays
HoldMaxRank
HoldMinDays
If
Index
InitPosition
Irate
Iterate
Last
LinearRegTrade
Log
LookbackDays
Lreg
MarginLoanFund
Max
Min
MMarket
Name
Not
Or
Penalty
PenaltyDays
PerShare
Portfolio
Positions
Pow
Print
RankInterval
RelativeRankSell
Roc
Rsi
RsiTrade

RubeCompatibility
Save
Select
Sell
Shares
Shift
Signal
SignalPairTrade
SlopeDays
SlopeFamilyTrade
SlopeFilter
SlopeRank
SlopeRankType
Sma
Sort
StartDate
Stdev
StopDate
Stretch
Sum
TaxLots
Total
Vector
Warp
WriteFile
ZigZag