

K and Real-time Data

This paper is intended to be a brief introduction to the real-time data facilities currently available in `k`. After reading this paper and working through the examples you should be able to incorporate real-time data into your `k` application.

`k` applications can currently access real-time market data through the Reuters Triarch network. At present, only record-based services are supported. The examples in this paper will use the services known as Marketfeed and Selectfeed. In order to run the examples, your `userid/workstation` name combination must be authorized to access the Triarch network. If you need help with this, consult your system administrator, or as a last resort, contact the `k` group (email `k-group`).

A Quick Demonstration

The basic utility library needed to access real-time data is the `md` library, located at `/ubs/itsu/opt/md/1.0/md.k` (or `/ubs/k/md.k`). Complete documentation of the functionality in this package is available in the file `/ubs/itsu/opt/md/1.0/md.doc` (Appendix B of this paper).

```
rousseau[nnyror]65: k
K(12/7)Copyright (c) 1993-1994 Atlantis Software, All Rights Reserved.
      Copyright (c) 1994-1995 Union Bank of Switzerland, All Rights Reserved.
Control-C to interrupt      \ to list commands      \\ to exit
```

```
\l /ubs/k/md
\v .
k t md
```

Note that the `md` directory now exists. It contains the functions we need. The first thing to do is run `.md.client[]`. This function announces our intention to subscribe to real-time data and will instantiate the real-time directory, `.R`, in our process. It will fail if we do not have access to the Triarch network. It is not strictly necessary to run this function first, but it is the best way to check for access to Triarch.

```
.md.client[]
(result is _n)
```

The best way to perform this check in an application is under error-trap protection:

```
@[.md.client;_n;:]
(0;)
```

Now let's subscribe to data on the March 1996 U.S. Treasury Bond futures contract from the Chicago Board of Trade. The information on this, and most other futures contracts, is available via the Marketfeed service. A quick check of the handles in `.R`, the real-time directory, shows that Marketfeed is indeed available.

```
!'.R
`EUROBROKER `GW_CDBTST `GOBT `IDN_MARKETFEED `LIBERTY `GARBAN `TULLET `HILL
`IPS `DDS `KRI `MLCS `DDSDEV `TELI `KNIGHT_RIDDER_PAGE `TRADITION `GW_CDBREC
`GW_CSRVREC `PATRIOT `RMJ `CHAPDELAIN `IDN_SELECTFEED `TELERATE `CANTOR `PRE-
BON `k_test `k_test1 `k_test2 `TELERATE_2 `TIPS_PAGE `TIPS_REC
```

Note that currently, IDN_MARKETFEED has no subscriptions:

```
!'.R.IDN_MARKETFEED
()
```

To subscribe to data, the function `.md.sub[src;item]` is run. `src` is the service name, in this case `'IDN_MARKETFEED`. `item` is the name of the instrument, according to Triarch. (Instrument names are commonly referred to as "rics"). The ric of the March 96 Bond contract is `'USH6`.

```
.md.sub['IDN_MARKETFEED;'USH6]
`USH6
```

The result of `.md.sub` is a symbol, which is the name of the subtree of the service directory which will now receive real-time updates from Triarch. The result will not always be the same as the item, since rics sometimes have symbols in them that are illegal in `k` identifiers. For instance, the ric for the current 30 year U.S. Treasury bond from GOVPX is `US30YT=PX`. Since `=` is an illegal character in `k` identifiers, the result of `.md.sub['IDN_SELECTFEED;'US30YT=PX']` (and the name of the subtree) will be `'US30YT_PX`.

Let's also subscribe to the Ten Year Note contract and the Five Year Note contract from the same source.

```
.md.sub['IDN_MARKETFEED;']`TYH6`FVH6
`TYH6 `FVH6
!'.R.IDN_MARKETFEED
`USH6 `TYH6 `FVH6
```

Triarch supplies a large amount of data about each instrument.

```
!'.R.IDN_MARKETFEED.USH6
`PROD_CATG `RDNDISPLAY `DSPLY_NAME `RDN_EXCHID `TIMACT `TRDPRC_1 `TRDPRC_2
`TRDPRC_3 `TRDPRC_4 `TRDPRC_5 `NETCHNG_1 `HIGH_1 `LOW_1 `PRCTCK_1 `CURRENCY
`TRADE_DATE `ACTIV_DATE `TRDTIM_1 `HST_CLOSE `BID `ASK `NEWS `NEWS_TIME `BID-
SIZE `ASKSIZE `ACVOL_1 `CONTR_MNTH `OPEN1 `OPEN2 `OPNRNGTP `CLOSE1 `CLOSE2
`CLSRNGTP `TRD_UNITS `LOTSZUNITS `LOT_SIZE `PCTCHNG `LOCHIGH `LOCLOW `OPINT_1
`OPINTNC `EXPIR_DATE `SETTLE `UPLIMIT `LOLIMIT `NUM_MOVES `OFFCL_CODE `HSTCLS-
DATE `LIMIT_IND `TURNOVER `BOND_TYPE `BCKGRNDPAG `YCHIGH_IND `YCLOW_IND
`PRC_QL2 `MKT_ST_IND `TRDVOL_1 `HIGHTP_1 `LOWTP_1 `LOT_SIZE_A `RECORDTYPE
`ACT_TP_1 `ACT_TP_2 `ACT_TP_3 `ACT_TP_4 `ACT_TP_5 `SEC_ACT_1 `SC_ACT_TP1 `SET-
TLEDATE `VOL_FLAG `IRGPRC `IRGVOL `IRGCOND `TIMCOR `SALTIM `TNOVER_SC `HST_VOL
`SESS_HIPLG `SESS_LOPLG `SSPRNG1 `SSPRNG2 `SSPRNGTP `RSMRNG1 `RSMRNG2 `RSM-
RNGTP `VOL_DATE `PRIMACT_1 `PRIMACT_2 `PRIMACT_3 `PRIMACT_4 `PRIMACT_5
`BCAST_REF `PRV_HIGH `PRV_LOW `CROSS_SC `OFF_CD_IND `SEQNUM `PRNTBCK `F1067
`F1078 `F1080 `F1352 `F1379 `F1383
```


For our quick demo, let's put up a small screen showing the ric, name, price, time of last update, and change on the day, for each instrument we have subscribed to.

```
\d app
rics..d:"!.R.IDN_MARKETFEED"           / ric of the instrument
names..d:".R.IDN_MARKETFEED[;`DSPLY_NAME]" / name of the instrument
prices..d:".R.IDN_MARKETFEED[;`TRDPRC_1]" / last trade price
times..d:".R.IDN_MARKETFEED[;`TRDTIM_1]" / last trade time
chgs..d:".R.IDN_MARKETFEED[;`NETCHNG_1]" / net change on the day
\d ^
`show$`app
`app
```

.kapp: 3						
rics	names			prices	times	chgs
USH6	US	T	BONDS Mar6	121.5625	17:40	0.21875
TYH6	US	T	NOTES Mar6	114.6562	17:36	0.15625
FVH6	US	T	NOTES Mar6	110.4688	17:36	0.109375

One important point to keep in mind when programming applications using these facilities is the *asynchronous* nature of the processes involved. If the code executed thus far was gathered up into a function and executed, the function would fail.

```
rousseau[nnyror]73: k
K(12/7)Copyright (c) 1993-1994 Atlantis Software, All Rights Reserved.
      Copyright (c) 1994-1995 Union Bank of Switzerland, All Rights Reserved.
Control-C to interrupt      \ to list commands      \\ to exit
```

```
\l /ubs/k/a/rt/rtdemo1.k
rt
{
  .md.client[]
  .md.sub[`IDN_MARKETFEED;]``USH6`TYH6`FVH6
  .k.app.rics..d:"!.R.IDN_MARKETFEED"
  .k.app.names..d:".R.IDN_MARKETFEED[;`DSPLY_NAME]"
  .k.app.prices..d:".R.IDN_MARKETFEED[;`TRDPRC_1]"
  .k.app.times..d:".R.IDN_MARKETFEED[;`TRDTIM_1]"
  .k.app.chgs..d:".R.IDN_MARKETFEED[;`NETCHNG_1]"
  `show$`.k.app
}
\e 1
rt[]
rank error
{.R.IDN_MARKETFEED[;`DSPLY_NAME]}
^
```

This happens because the code executes faster than the time needed for the data to flow into the real-time directory. When k tries to evaluate the dependency definition for names, there are as yet no items in the subdirectories:

```
> !'.R.IDN_MARKETFEED.USH6
()
```

So, while what we've done so far is fine for learning and experimentation, another approach is required for real applications. Before tackling the problem of asynchronous programming, let's examine how to monitor the status of the connection between *κ* and Triarch.

The MD Attribute Directory

As shown above, the real-time directory, *.R*, is instantiated when the function *.md.client[]* is run. The subdirectories of *.R* are the available Triarch services, such as *'IDN_MARKETFEED*. Each of these subdirectories has a special attribute directory named *MD* which always contains three entries: *status*, *text* and *type*.

```
!'.R.IDN_MARKETFEED.
, 'MD
!'.R.IDN_MARKETFEED..MD
`type `status `text
.R.IDN_MARKETFEED..MD
.((`status;`valid;)
(`type;`record;)
(`text;"";))
```

The *type* attribute will have the value *'record*, *'page* or *'unknown*. The latter value is a transient condition possible during a brief period following the execution of *.md.client[]*. The *status* attribute will have the value *'valid* when the service is operating normally and delivering updates, *'invalid* when there is a temporary interruption of service and *'closed* when the service has shut down. As with the *type* attribute, the *status* attribute may also have a value of *'unknown* just after the execution of *.md.client[]*. The *text* attribute may contain a string elaborating further on the status of a service. The default value of *text* is *""*. One way to utilize this information in an application is to put a trigger on the *status* attribute. When *status* changes, your trigger code can take appropriate action, such as informing the user, or attempting to reconnect (in case the service becomes *'closed*).

An *MD* directory also exists one level down from the service subdirectories; that is, at the *ric* level.

```
.R.IDN_MARKETFEED.USH6..MD
.((`status;`valid;)
(`text;"";))
```

Notice that there is no *type* attribute at this level. Triggers can also be used here to monitor the status of individual subscriptions, or, the value of *status* can be displayed in a table along with other real-time data.


```
\d app
status..d:".R.IDN_MARKETFEED[~!.R.IDN_MARKETFEED;`MD;`status]"
```

.kapp:3							
rics	names			prices	times	chgs	status
USH6	US	T	BONDS Mar6	121.625	17:47	0.28125	valid
TYH6	US	T	NOTES Mar6	114.6875	17:47	0.1875	valid
FVH6	US	T	NOTES Mar6	110.4688	17:36	0.109375	valid

Triggers and Asynchronous Programming

Triggers on the `status` attribute of the MD directory can be used to solve the above-mentioned problem of the program running faster than the subscription process. The idea here is to preset a trigger on the status attribute of each ric in the real-time dictionary before subscribing. The trigger displays the screen when the status of all the rics are `valid`. Here's the complete script followed by a sample session where the code is run:

```
\l /ubs/k/md

\d .rt

Feed:`.R.IDN_MARKETFEED

app.rics..d:".R.IDN_MARKETFEED"
app.names..d:".R.IDN_MARKETFEED[;`DSPLY_NAME]"
app.prices..d:".R.IDN_MARKETFEED[;`TRDPRC_1]"
app.times..d:".R.IDN_MARKETFEED[;`TRDTIM_1]"
app.chgs..d:".R.IDN_MARKETFEED[;`NETCHNG_1]"

demo:{
  .md.client[]
  rics:`USH6`TYH6`FVH6
  paths:~`$(((($Feed),"."),/:(($rics)),\:"."),\:".MD.status"
  trigs:(".rt.trig[\\"",/:(($rics)),\:".\"")
  paths .[;`t;:]'trigs
  .md.sub[`.IDN_MARKETFEED;]'rics
}

trig: {[ric]/trig
  `0:ric," ",($._v)), "\n"
  if[~&/`valid~/:Feed[~!Feed;`MD;`status];:_n
  `show$.rt.app
}
```

/setup .R
/rics we want
/paths to triggers
/triggers
/set triggers
/subscribe

/which one got updated
/if any are not valid, exit
/otherwise, show the screen

```
rousseau[nnyror]81: k
K(12/7)Copyright (c) 1993-1994 Atlantis Software, All Rights Reserved.
      Copyright (c) 1994-1995 Union Bank of Switzerland, All Rights Reserved.
Control-C to interrupt      \ to list commands      \\ to exit
```

```
\l /ubs/k/a/rtdemo2
.rtdemo[]
USH6 pending
TYH6 pending
FVH6 pending
USH6 valid
TYH6 valid
FVH6 valid
/at this point, the screen pops up
```

U.S. Treasury Bills, Notes & Bonds

In order to receive, calculate and display real-time pricing data on U.S. Treasury securities, several pieces of information must be known. First the Reuters identifier, or ric, must be determined. Next, the indicative information for the securities to be monitored must be retrieved from a securities database. Finally, the subscription for real-time data must be made, and the appropriate triggers put in place. For the following example, we will retrieve GOVPX data from the Selectfeed service and use indicative data from the EJV databases. We will set up a screen that displays the current price/discount rate and bond equivalent yield for the current U.S. On-The-Run Treasuries, including the 3, 6 and 12 month bills, the 2, 3, 5 and 10 year notes, and the 30 year bond. All of the code for this example is located in the script /ubs/k/a/rt/rtdemo3.k.

First we need to load several utilities:

```
\l /ubs/k/md          / real-time utilities
\l /ubs/k/i/servers    / k data server utilities
\l /ubs/k/f/b          / price-yield analytics
\l /ubs/k/a/utls       / miscellaneous utilities
```

Next, we ask the EJV data server for a list of the current On-The-Run cusips:

```
otr:*|.i.sync[`ejv_mike;(`otr;)]
otr
`"912794X7" ` "912794Z4" ` "9127942B" ` "912827V9" ` "912827V7" ` "912827W2"
`"912827V8" ` "912810EV"
```

Then we retrieve the indicative data for those securities:

```
!data:*|.i.sync[`ejv_mike;(`bonds;,$otr)]
`amt_iss_tot `amt_outsd `asset_id `asset_status_cd `asset_tmpltd_cd
`call_sched_fl `cpn_class_cd `cpn_rate `cpn_type_cd `cusip `dated_dt
`days_to_settle `day_cnt_cd `eom_pmt_fl `er_score `fig_rule_cd `first_cpn_dt
`fund_sched_fl `iss_dt `iss_price `iss_yld `last_cpn_dt `mat_dt `mat_type_cd
`native_yld_type_cd `orig_spread `pmt_freq_cd `pricing_bmk_id `put_sched_fl
`rate_sched_fl `redemption_value `standard_yld_type_cd `ticker `call_sched
`fund_sched `fund_terms `put_sched `rate_sched
```


data is a dictionary of lists. In order to proceed with the example, it must be transformed into a list of dictionaries.

```
data:.ul.dv2vd[data] / dictionary of vectors to vector of dictionaries
#data / so now there are 8 dictionaries
8
!data[0] / and each one has the same handles as the original
`amt_iss_tot `amt_outsd `asset_id `asset_status_cd `asset_tmplt_cd
`call_sched_fl `cpn_class_cd `cpn_rate `cpn_type_cd `cusip `dated_dt
`days_to_settle `day_cnt_cd `eom_pmt_fl `er_score `fig_rule_cd `first_cpn_dt
`fund_sched_fl `iss_dt `iss_price `iss_yld `last_cpn_dt `mat_dt `mat_type_cd
`native_yld_type_cd `orig_spread `pmt_freq_cd `pricing_bmk_id `put_sched_fl
`rate_sched_fl `redemption_value `standard_yld_type_cd `ticker `call_sched
`fund_sched `fund_terms `put_sched `rate_sched
```

Lastly, we need the rics of the issues in order to subscribe to the Triarch data:

```
#allcusips:*.i.sync[`ejv_mike;(`Cusips;)] / retrieve all the cusips
248
#allrics:*.i.sync[`ejv_mike;(`pxrics;)] / retrieve all the rics
248
rics:allrics@&allcusips _lin otr / keep just the ones we want
rics
`"US321T=PX" ` "US620T=PX" ` "USD12T=PX" ` "US53/N97T=PX" ` "US54/N98T=PX"
`"US55/N00T=PX" ` "US57/N05T=PX" ` "US67/825T=PX"
```

Now we have the data required. The remaining steps are to subscribe to the real-time data and set up a trigger which will cause the appropriate calculations to run whenever a new price arrives from Triarch. One way to do this is to write a cover function to initialize the portions of the real-time directory required, set the trigger, and request the subscription from Triarch. Here is a sample function to do this:

```
.rt.subscribe:[src;ric;data]/subscribe
/subscribes from src, to ric, with indicative info data; returns id
fna:-999999999.0 /floating pt na
dir:`$.R.", $src /real-time dictionary
id:.md.sub[src;ric] /subscribe to real-time data
.[dir;(id;`Ric);::ric] /assign security ric
.[dir;(id;`Yield);::fna] /initialize yield
.[dir;(id;`Data);::data] /set indicative data dict.
.[dir;(id;`TRDPRC_1);::fna] /initialize trade price
.[dir;(id;`TRDTIM_1);::"] /initialize trade time
.[dir;(id;`DSPLY_NAME);::"] /initialize display name
.[dir;(id;`t);::".rt.update[_v;]*_i;"] /set trigger
id}
```

Let's examine what subscribe is doing in detail. The first argument, src, is the name of the service from which we are subscribing. The two services we are using in these examples are 'IDN_MARKETFEED and 'IDN_SELECTFEED. The second argument, ric, is the previously described Triarch security identifier. The last argument, data, is a dictionary of indicative data for the security being subscribed to. This includes the standard info needed to run price/yield cal-

culations on Fixed Income securities, such as coupon rate and maturity date. The main idea of this function is to add a few extra variables to the real-time data structure. (Since Triarch fields are always all uppercase, a useful convention when adding application-specific information fields to these directories is the use of mixed case names.) These additional fields will hold both extra information needed by our application, as well as results generated by the analytics.

Note that the trigger is set on the entire ric subdirectory. This means that the trigger will run whenever any field in the directory is updated by the Triarch subscription. The trigger function, `update`, is passed the name of the subdirectory (`_v`) and is invoked once for each field updated (`_i`) by way of an "each" expression. Using an each expression simplifies the logic of the trigger function, since only one field will be handled at a time.

Here is the update trigger function:

```
.rt.update: {[v;i]/update
  /trigger for securities receiving data from triarch
  if[i~`TRDPRC_1
    res:calc[v[`TRDPRC_1];*v[`Data];`Price;*_ltime _t]
    @[v;`Yield;::res[`yield]]
  }
```

The update function first checks the value of `i` to see if the trade price, `TRDPRC_1`, has been updated. If not, update ends. If `TRDPRC_1` is the changed field, the price/yield analytic, `calc`, is run, and the results are saved back into the active subdirectory.

Another alternative is to monitor not only the `TRDPRC_1` field, but also the `BID` and `ASK` fields. Typically, a new field called `Qt` (for "quote") is defined as either the bid/ask midpoint, or the last trade price. If either `BID` or `ASK` is empty, `Qt` remains unchanged. Otherwise `Qt` is set to the average, and the price/yield analytic is rerun. Calculations can also be run for the individual `BID` and `ASK` numbers, and the times can be saved. The code to do all of that looks like this:

```
.rt.altupdate: {[v;i]/alternative update function
  if[i _in `BID`ASK`TRDPRC_1
    if[|/b:`BID`ASK _lin ,i /if we got bid or ask,
      res:calc[v[i];*v[`Data];`Price;Trade] /run the calcs
      @[v;*(`BidYield`BidTime;`AskYield`AskTime)@&b;::res[`yield],_t]
      if[~|/v[`BID`ASK] _lin DNA;@[v;`Qt`QtTime;::(.5*+/v[`BID`ASK]),_t]]
    if[i~`TRDPRC_1 /if we got trade, set Qt to it
      @[v;`Qt`QtTime;::v[i],_t]]
    res:calc[v[`Qt];*v[`Data];`Price;Trade] /always run calcs for Qt
    @[v;`Yield`MDur`PV01;::res[`yield],res[`risks`,`mduration`pv01]]
  }
```

The examples here use `.rt.update`.

The calculation function looks like this:


```
.rt.calc: {[quote;data;type;trade]/calc
/get yield given price or discount quote
/data is indicative data
/type is `Yield or `Price (if `Price, may be changed to `Discount)
/trade is trade date
if[~type=`Yield;type:*(`Price`Discount)&&data[`asset_tmpltd]=`TC`TD]
req:.*((`given `rate `calc_risks `deal_date);(type;quote;1;trade))
res:.b.calc_rslt[req;data]
if[~0=res[`stat];`res[`mesg]]
res[`outp]}
```

Now let's put it all together. One way to start is to create a template directory to hold a "proto-type" of the screen variables.

```
\d .rt.setupOne          /template for basic calculator

Ric..d: ".R.IDN_SELECTFEED"
Status..d: ".R.IDN_SELECTFEED[~!.R.IDN_SELECTFEED;`MD;`status]"
Time..d: ".R.IDN_SELECTFEED[;`TRDTIM_1]"
Time..f: `.ul.tsFmt
Name..d: ".R.IDN_SELECTFEED[;`DSPLY_NAME]"
Name..f: 16$
Price..d: ".R.IDN_SELECTFEED[;`TRDPRC_1]"
Price..f: 10.6$
Yield..d: ".R.IDN_SELECTFEED[;`Yield]"
Yield..f: 7.3$
Yield..l: "Yield"
```

We also need a set up function which will perform all the preliminary steps outlined above:

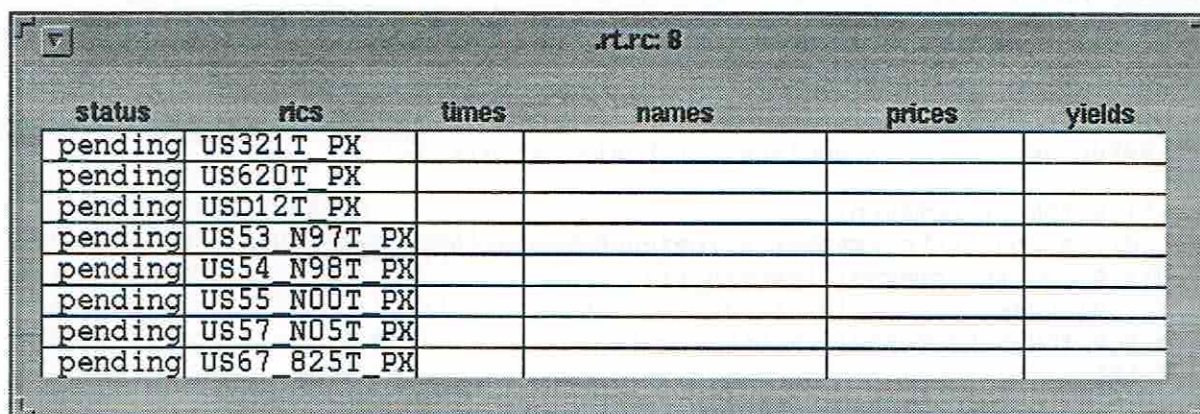
```
.rt.setup: {[]/setup
  if[*@[.md.client;_n;:]
    .ul.suicideNote["You Do NOT Have Access To Triarch Real-Time Data";10]
    :_n]
  otr:*|.i.sync[`ejv_mike;(`otr;)]           /suicideNote is an alertbox
  data:*|.i.sync[`ejv_mike;(`bonds;,$otr)]    /on-the-run cusips
  data:.ul.dv2vd[data]                       /and indicative data
  allcusips:*|.i.sync[`ejv_mike;(`Cusips;)]   /vector of dictionaries
  allrics:*|.i.sync[`ejv_mike;(`pxrics;)]     /all the cusips
  rics:allrics&&allcusips _lin otr            /all the rics
  rics subscribe[`IDN_SELECTFEED;]'data       /just the ones we want
                                              /subscribe to the rics
}
```

Then we need a function to run the set up function, instantiate a copy of the screen variables, and show them:

```
.rt.demol: {[]/demol
  setup[]          /run the setup function
  rc1:.rt.setupOne /set screen vars
  `show$.rt.rc1    /show the screen
}
```

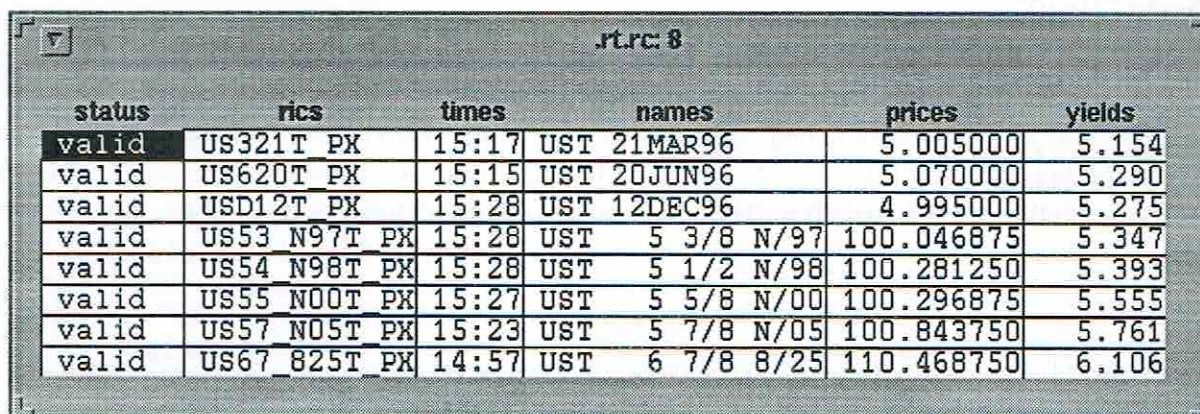

The first thing you should be wondering is why this function will work, when the previous demo function wouldn't, until we changed the logic to examine the 'status' attribute before attempting to show the screen. Good question! (If you didn't wonder about that, then you haven't been paying attention). In this example, the `subscribe` function initializes all the fields we will be displaying. This allows us to show the screen even before the fields have Triarch supplied values. The initial display of the screen will simply show the values used as defaults in `subscribe`.

So, here's what the screen looks like when it first comes up:



status	rics	times	names	prices	yields
pending	US321T PX				
pending	US620T PX				
pending	USD12T PX				
pending	US53 N97T PX				
pending	US54 N98T PX				
pending	US55 N00T PX				
pending	US57 N05T PX				
pending	US67_825T PX				

A few seconds later, here's what we see:



status	rics	times	names	prices	yields
valid	US321T PX	15:17	UST 21MAR96	5.005000	5.154
valid	US620T PX	15:15	UST 20JUN96	5.070000	5.290
valid	USD12T PX	15:28	UST 12DEC96	4.995000	5.275
valid	US53 N97T PX	15:28	UST 5 3/8 N/97	100.046875	5.347
valid	US54 N98T PX	15:28	UST 5 1/2 N/98	100.281250	5.393
valid	US55 N00T PX	15:27	UST 5 5/8 N/00	100.296875	5.555
valid	US57 N05T PX	15:23	UST 5 7/8 N/05	100.843750	5.761
valid	US67_825T PX	14:57	UST 6 7/8 8/25	110.468750	6.106

You may have noticed that the `subscribe` function does not initialize the `status` field. It doesn't have to. The `.md.sub` function initializes it for us.

Who's In Control?

Occasionally, you may need to write an application in which fields on the screen may be updated by both user input and real-time data. For example, a bond price/yield calculator may display real-time data but allow the user to enter an arbitrary price or yield to be run through the price/yield analytic. The problem with this kind of application is that you never know when real-time data is going to show up. The user might be typing a price into a field, and right in the middle of his typing, the field is completely overwritten by the arrival of new real-time data. Clearly what is

needed is a user controlled switch which toggles the feed on and off. The implementation of such a switch is straightforward. Rather than actually turning the real-time feed on and off, we can simply add a flag variable which the various screen dependency definitions test. We can also add a checkbox to the screen which shows the state of the flag variable, and flips that state when pressed. It's also a nice idea to set the editable state attribute on the appropriate fields on the screen depending on whether the feed is "on" or "off". The main idea is that real-time data continues to flow into the real-time directory, .R, but the screen variables are only updated when it is appropriate to do so.

Here's the template for enhanced screen:

```
\d .rt.setupTwo          /template for Mode-controlled calculator

Ric..d:"!.R.IDN_SELECTFEED"
Status..d:".R.IDN_SELECTFEED[~!.R.IDN_SELECTFEED;`MD;`status]"
Time..d:"[.rt.Mode;.R.IDN_SELECTFEED[;`TRDTIM_1];._v]"
Time..f:`ul.tsFmt
Name..d:".rt.Mode;.R.IDN_SELECTFEED[;`DSPLY_NAME];._v]"
Name..f:16$
Price..d:".rt.Mode;.R.IDN_SELECTFEED[;`TRDPRC_1];._v]"
Price..f:10.6$
Yield..d:".rt.Mode;.R.IDN_SELECTFEED[;`Yield];._v]"
Yield..f:7.3$
Yield..l:"Yield"
```

As you can see, the dependency definitions for the Time, Name, Price and Yield fields have been changed to an if-then-else expression where the condition is the value of .rt.Mode. These fields will be invalidated if either .rt.Mode changes, or if .R.IDN_SELECTFEED changes. If .rt.Mode is 0, the value of the field will remain the same (._v). Now let's examine the new main function:

```
.rt.demo2: {[ ]/demo2
  setup[ ]                                /run the setup function
  .rt.Mode:1                             /feed; 0: off, 1: on
  .rt.Mode..c:`check                      /make it a check box
  .rt.Mode..l..d:".rt.Mode;\\"Feed is ON\\";\\\\"Feed is OFF\\"]"
  .rt.Mode..t:".rt[`rc2;`Price.`Yield.;`e]:~Mode" /adjust edit mode
  rc2::rt.setupTwo                       /set screen vars
  .rt.rc2..t:".rt.input[_v;_i]"           /trigger on directory
  .rt[`rc2;~!.rt.rc2;`e]:0               /protect all fields
  .rt..a::`Mode`rc2                      /arrangement
  `show$`.rt                             /show the screen
}
```

In addition to running the setup function and instantiating a copy of the screen variables, as before, this function also does a few other things. First, it creates the variable .rt.Mode, which will be the flag determining whether real-time data flows to the screen or not. As we saw, several fields on the screen depend on this variable. .rt.Mode will itself appear on the screen as a checkbox widget; the feed will be on when the checkbox is pushed in. The label for .rt.Mode also depends on its value. The trigger for .rt.Mode adjusts the editable state (protection) of the Price and Yield fields appropriately. Note also that all the fields in the table are initially set to be non-

editable. Another key difference is that this version of the demo shows the checkbox `Mode` as well as the screen directory `rc2` in the same form, (after setting the arrangement attribute `.a`) rather than just the screen directory. This allows the display of `.rt.Mode` in the same form. The final enhancement is the placement of a trigger on the screen directory, which comes into play when the real-time feed is off and the screen is used as a calculator. Here are shots of the screen with the feed on and off:

rt							
<input type="checkbox"/> Feed is ON							
rc2: 8							
Ric	Status	Time	Name	Price	Yield		
US328T_PX	valid	11:42	UST 28MAR96	4.940000	5.083		
US627T_PX	valid	11:44	UST 27JUN96	4.950000	5.159		
USD12T_PX	valid	11:58	UST 12DEC96	4.910000	5.176		
US52_D97T_PX	valid	11:57	UST 5 1/4 D/97	100.125000	5.183		
US54_N98T_PX	valid	11:51	UST 5 1/2 N/98	100.671875	5.243		
US54_D00T_PX	valid	11:56	UST 5 1/2 D/00	100.406250	5.406		
US57_N05T_PX	valid	11:56	UST 5 7/8 N/05	102.093750	5.595		
US67_825T_PX	valid	11:53	UST 6 7/8 8/25	112.578125	5.965		

rt							
<input type="checkbox"/> Feed is OFF							
rc2: 8							
Ric	Status	Time	Name	Price	Yield		
US328T_PX	valid	11:42	UST 28MAR96	4.940000	5.083		
US627T_PX	valid	11:44	UST 27JUN96	4.950000	5.159		
USD12T_PX	valid	11:58	UST 12DEC96	4.910000	5.176		
US52_D97T_PX	valid	12:00	UST 5 1/4 D/97	100.132812	5.179		
US54_N98T_PX	valid	12:01	UST 5 1/2 N/98	100.671875	5.243		
US54_D00T_PX	valid	12:01	UST 5 1/2 D/00	100.414062	5.404		
US57_N05T_PX	valid	11:59	UST 5 7/8 N/05	102.109375	5.593		
US67_825T_PX	valid	11:53	UST 6 7/8 8/25	112.578125	5.965		

Any change to the screen will fire the trigger. Since dependency invalidation does not fire triggers, the only times the trigger will run are when the editable attribute on the Price or Yield field changes (i.e. when the feed is toggled on or off), or when the feed is off and the user types a price or a yield into a cell of the table.

Here is the trigger function:

```
.rt.input: {[v;i]/input
  /compute price or yield based on user input
  if[~(*i)_in `Price `Yield;:_n]           /only calc for price or yield input
  data:Feed[(v@`Ric)*@|i;`Data]           /indicative data
  b:`Price=*i                             /used to determine types
```



```

c:data[asset_tmplt_cd]='TC           /used to determine result price type
inType:('Yield`Price)@b             /input type
outType:('Yield`Price)@~b           /output type
resType:(((drate`price)@c),`yield)@b /result type
res:calc[. [v;i];data;inType;*_ltime _t] /run the calc function
.[v;outType,*|i;;;res@resType]      /update the screen
}

```

Here are the values of the arguments, *v* and *i*, after entering a new price in one of the rows of the table:

```

> v
`.rt.rc2
> i
(`Price;5)

```

As you can see, all the information needed to run the price/yield calculator, and set the result back on the screen is available in, or can be derived from, these variables. One subtlety should be pointed out. The last line of `.rt.input` sets a cell of one of the fields in the table, which would normally fire the trigger. But since the trigger is already executing it is not fired again, which is the desired effect. When the feed is toggled back on, the screen dependencies will be invalidated (since they depend on `.rt.Mode`) and any numbers that have been typed onto the screen in calculation mode will be overwritten by the latest real-time data. Even if no new data arrived during the period in which the feed was off, the screen will still be updated with the most recent data in `.R`.

One useful rule-of-thumb which can be inferred from these examples is that when mixing real-time data with user-input data, dependencies should be used to handle the display of real-time data and triggers should be used to handle the user-input data. Another important point is that it is often more useful to put a trigger on the directory in which a variable exists, instead of directly on the variable itself.

Appendix A: Sample Scripts

The following four pages contain the three sample scripts mentioned in the paper. These scripts can be found on-line in the directory `/ubs/k/a/rt`.

/rtdemo1.k - companion script for the paper: K and Real-time Data

\l /ubs/k/md

\e 1

```
rt: {[]/rt - this function will fail with a rank error
.md.client[]
.md.sub['IDN_MARKETFEEED;'] 'USH6' TYH6'FVH6
.k.app.rics..d:".R.IDN_MARKETFEEED"
.k.app.names..d:".R.IDN_MARKETFEEED[; 'DSPLY_NAME]"
.k.app.prices..d:".R.IDN_MARKETFEEED[; 'TRDPRC_1]"
.k.app.times..d:".R.IDN_MARKETFEEED[; 'TRDTIM_1]"
.k.app.chgs..d:".R.IDN_MARKETFEEED[; 'NETCHNG_1]"
'show$.k.app
}
```

/rtdemo2.k - companion script to the paper: K and Real-time Data

\l /ubs/k/md

\e 1

\d .rt

Feed: '.R.IDN_MARKETFEED

app.rics..d: "!R.IDN_MARKETFEED"

app.names..d: ".R.IDN_MARKETFEED[; 'DSPLY_NAME]"

app.prices..d: ".R.IDN_MARKETFEED[; 'TRDPRC_1]"

app.times..d: ".R.IDN_MARKETFEED[; 'TRDTIM_1]"

app.chgs..d: ".R.IDN_MARKETFEED[; 'NETCHNG_1]"

demo:{

.md.client[]

rics: 'USH6' TYH6' FVH6

/setup .R

/rics we want

paths: ~'\$((((Feed), ".") ,/:(\$rics)), \: ".") , \: ".MD.status"

/paths to triggers

trigs: (" .rt.trig[\" ,/:(\$rics), \: "\"]"

/triggers

paths .[; 't; ;:] 'trigs

/set triggers

.md.sub['IDN_MARKETFEED;]'rics

/subscribe

}

trig: {[ric]/trig

'0:ric, " ", (\$(_v)), "\n"

/which one got updated

if[~&/'valid~/:Feed[~!Feed; 'MD; 'status];:_n]

/if any are not valid, exit

'show\$'.rt.app

/otherwise, show the screen

}

/rtdemo3 - companion script to the paper: K and Real-time Data

```
\l /ubs/k/md /real-time utilities
\l /ubs/k/i/servers /k data server utilities
\l /ubs/k/f/b /price-yield analytics
\l /u/mkr/k/a/utls /miscellaneous utilities
```

\e 1

```
\d .rt.setupOne /template for basic calculator
```

```
Ric..d:"!.R.IDN_SELECTFEED"
Status..d:".R.IDN_SELECTFEED[~!.R.IDN_SELECTFEED;'MD;'status]"
Time..d:".R.IDN_SELECTFEED['TRDTIM_1]"
Time..f:".ul.tsFmt"
Name..d:".R.IDN_SELECTFEED['DSPLY_NAME]"
Name..f:16$
Price..d:".R.IDN_SELECTFEED['TRDPRC_1]"
Price..f:10.6$
Yield..d:".R.IDN_SELECTFEED['Yield]"
Yield..f:7.3$
Yield..l:"Yield"
```

```
\d .rt.setupTwo /template for Mode-controlled calculator
```

```
Ric..d:"!.R.IDN_SELECTFEED"
Status..d:".R.IDN_SELECTFEED[~!.R.IDN_SELECTFEED;'MD;'status]"
Time..d:". [.rt.Mode;.R.IDN_SELECTFEED['TRDTIM_1'];._v]"
Time..f:".ul.tsFmt"
Name..d:". [.rt.Mode;.R.IDN_SELECTFEED['DSPLY_NAME'];._v]"
Name..f:16$
Price..d:". [.rt.Mode;.R.IDN_SELECTFEED['TRDPRC_1'];._v]"
Price..f:10.6$
Yield..d:". [.rt.Mode;.R.IDN_SELECTFEED['Yield'];._v]"
Yield..f:7.3$
Yield..l:"Yield"
```

\d .rt

FNA:-999999999.0

/floating point NA value

Feed:!.R.IDN_SELECTFEED

/real-time directory

```
calc:[quote;data;type;trade]/calc
/get yield given price or discount quote
/data is indicative data
/type is 'Yield or 'Price (if 'Price, may be changed to 'Discount)
/trade is trade date
if[~type='Yield;type:*('Price'Discount)&&data['asset_tmpltd']='TC'TD]
req:+(('given'rate'calc_risks'deal_date);(type;quote;1;trade))
res:.b.calc_rslt[req;data]
if[~0=res['stat'];res['mesg']]
res['outp]]
```

```
demo1:[[]/demo1
  setup[] /run the setup function
  rc1::rt.setupOne /set screen vars
  'show$.rt.rc1 /show the screen
]
```

```
demo2:[[]/demo2
  setup[] /run the setup function
  .rt.Mode:1 /feed; 0 is off, 1 is on
  .rt.Mode..c:'check /make it a check box
  .rt.Mode..l..d:". [.rt.Mode;"Feed is ON";"Feed is OFF"]" /label depends on state
  .rt.Mode..t:".rt['rc2;'Price.'Yield.;'e'];~Mode" /adjust edit mode
```

```

rc2::rt.setupTwo
    .rt.rc2..t:".rt.input[_v;_i]"
    .rt['rc2;~!.rt.rc2;'e]:0
    .rt..a:.'Mode'rc2
    'show$'.rt
}

input: {[v;i]/input
    /compute price or yield based on user input
    \*in trigger"
    \v
    \i
    if[~(*)_in 'Price 'Yield;:_n]
    data:Feed[(v@'Ric)@*|i;'Data]
    b:'Price=*i
    c:data['asset_tmpl_t_cd]='TC
    inType:('Yield'Price)@b
    outType:('Yield'Price)@~b
    resType:(((('drate'price)@c), 'yield)@b
    res:calc[. [v;i];data;inType;*_ltime _t]
    . [v;outType,*|i;::res@resType]
}

setup: {[]/setup
    if[*@['.md.client;_n;:]
        .ul.suicideNote["You Do NOT Have Access To Reuters Triarch Real-Time Data";10]
        :_n]
    otr:*|.i.sync['ejv_mike;('otr;)]
    data:*|.i.sync['ejv_mike;('bonds;,$otr)]
    data:.ul.dv2vd[data]
    allcusips:*|.i.sync['ejv_mike;('Cusips;)]
    allrics:*|.i.sync['ejv_mike;('pxrics;)]
    rics:allrics@allcusips _lin otr
    rics subscribe['IDN_SELECTFEED;]'data
}

subscribe: {[src;ric;data]/subscribe
    /subscribes from src, to ric, with indicative info data; returns id
    dir:'$.R.", $src
    id:.md.sub[src;ric]
    . [dir;(id;'Ric);::ric]
    . [dir;(id;'Id);::id]
    . [dir;(id;'Yield);::FNA]
    . [dir;(id;'Data);::data]
    . [dir;(id;'TRDPRC_1);::FNA]
    . [dir;(id;'TRDTIM_1);::"]
    . [dir;(id;'DSPLY_NAME);::"]
    . [dir;(id;'t);::".rt.update[_v;]*_i;"]
    id}

update: {[v;i]/update
    /trigger for securities receiving data from triarch
    if[i~'TRDPRC_1
        res:calc[v['TRDPRC_1]*v['Data;'Price;*_ltime _t]
        @[v;'Yield;::res['yield]]
    }

```


Appendix B: MD Documentation

This description of the basic MD functionality is an excerpt from the document `/ubs/itsu/opt/md/1.0/md.doc`.

Except where otherwise noted, all functions return `nil` (`_n`) in the absence of errors. Errors are signalled.

`sub[src; item]`

Subscribe to the data item identified by the symbols `src` and `item`. If this is the first reference to the item, or if a previous subscription to it has been ``closed`, the item status is set to ``pending`; otherwise status is unchanged, but an additional image may be sent, updating all fields at once. This function returns a symbol denoting the base name of the entry in the `.R` directory to which data will be delivered; ordinarily this will be the same as `item` unless `item` contains characters which are illegal in `K` directory entries, in which case the name will be suitably modified; often this simply means substituting underscore `_` for equals `=`.

`unsub[src; item]`

Close the subscription to the data item identified by `src` and `item`. The item status is set to ``closed` and updates cease (though they may not cease immediately).

`client[]`

Announce yourself as a subscription client. This is done automatically by `sub` and is not ordinarily necessary but can be used to permit tracking of global status information in the `.R` directory without actually establishing any subscriptions, or to install information about available sources into the `.R` directory.

`serve[src; server; type; ncache; cb]`

Announce yourself as a source provider for the service named by the symbol `src`. `server` is a symbol identifying the name of the service on the local network (in many cases this will be the same as `src`). `type` is a symbol identifying the type of the records to be served, and (currently) must be ``record`. `ncache` is an integer specifying the maximum number of items that you are prepared to serve; it may be constrained by per-source configuration limits. `cb` is a callback function (or symbol reference to same) that is invoked whenever a subscription request is received, as in `cb[src; item; tag]` where `src` and `item` are symbols denoting the source and item being requested, and `tag` is an integer ID which must be supplied as an argument to the `wimage` or `wrefuse` with which the request is satisfied or denied.

`unserve[src]`

Terminate service for source `src`. Subscribers will see a status message that will set the status of all subscriptions for that source to ``closed`.

`wimage[src; item; tag; data]`

Send a record image for the source and item denoted by `src` and `item`. For the initial image written in response to a subscription request, the integer `tag` must match the tag supplied with the request; any subsequent images should supply a value of 0 for the tag. The data is in the

form of a dictionary whose entries are IDN field names (as described above) and whose values are the corresponding field values. Every image sent must include all fields in the record; it is not possible to send a partial set of fields in an image and augment that set in a subsequent update. A new image must be sent in order to add fields to a record. (There is currently no way to delete fields from a record.)

wrefuse[src; item; tag; text]

Refuse a subscription request for source `src` and item `item` with tag `tag`. A character string describing the reason for the denial can be supplied in `text`; alternatively `text` can contain the integer value 0 to send no text description, or the integer value 1 to send a system-defined default message.

wupdate[src; item; data]

Send an update for the item denoted by `src` and `item`. The dictionary data is formatted as described for `wimage` but contains only the fields that are being updated. The first update should not be written until the initial image has been written.

wclose[src; item; text]

Close the subscription for the item identified by `src` and `item`. A character string describing the reason for closing can be supplied in `text`; alternatively `text` can contain integer 0 to send no text, or integer 1 to send a system-defined default message.

wstatus[src; item; status; text]

Send a status message for the item denoted by `src` and `item`. `status` must be either ``valid` or ``invalid`. `text` can contain a character string (which may be empty) describing the status, or integer 0 to send no text, or integer 1 to send a system-defined default message. A status message for an item can be sent at any time after the initial subscription request, and in particular can be sent before the initial image. A ``valid` status message for an item should be sent prior to the image if there will be any significant delay in sending the image, to prevent the subscription request from being timed out.

wgstatus[src; status; text]

Send a global status message for source `src`, pertaining to all items. `status` must be ``valid` or ``invalid`. `text` can contain a character string (which may be empty) describing the status, or integer 0 to send no text, or integer 1 to send a system-defined default message. If all source service is to be temporarily interrupted, an ``invalid` global status message should be issued, followed by a ``valid` global status message when service is resumed.

idnadd[fname; fid; ftype]

Add, to the map of known record field names, a new name denoted by the symbol `fname`, with integer `fid` (which must be less than 2048) and symbol `ftype` (which must be one of ``ALPHA-NUMERIC`, ``ALPHANUM_XTND`, ``BINARY`, ``ENUMERATED`, ``INTEGER`, ``NUMERIC`, ``PRICE`, ``DATE`, ``TIME` or ``TIME_SECONDS`). If a client and server wish to define and exchange new fields they must both add them with `idnmap`. This interface is Reuters-specific and is likely to change in the future.