APPENDIX G

32 SHEETS

```
#!/usr/local/bin/tclsh-misc
#
#  Part of payment system prototype
#
#   This is the CGI script that takes URLs, processes them as payment orders,
#   and returns a redirect to the real URL.
#
#  Andrew Payne
#  payne@openmarket.com
#


#  Load support routines.
#
set library {../lib}
source $library/mall.conf
source $library/log.tcl
source $library/database.tcl
source $library/ticket.tcl
source $library/cgilib.tcl
source $library/payment.tcl

run_with_log {

sys!cginit nph-pi.cgi pid

# open database for read-write use
open_database


# Write out the details of the payment order, in HTML  (this routine is
# used in the various {return-*} routines)
#
proc payment_details {} {
    global merchant fields
set msg [<p><blockquote><b>Merchant:</b> $merchant(principal_name)<br>
<b>Description:</b> $fields(desc)<br>
<b>Amount:</b> $fields(amt) ($fields(cc) currency)<br>
</blockquote><p>
]
    return [subst $msg]
}


#
# Write out the details of the payment order, in HTML  (this routine is
# used in the various {return-*} routines)
#
proc payment_details_pre {} {
    global merchant fields
set msg {Merchant: $merchant(principal_name)
Description: $fields(desc)
Amount: $fields(amt) ($fields(cc) currency)
}
    return [subst $msg]
}

# ------------------------------------------------------------------------
#
#  Return an pre-confirmation page to the client.  This page is returned
#  before we know the identity of the user, and gives the user the
#  opportunity to establish an account if they don't already have one.
#
# new_account_link and demo_details are imported from mall.conf
#
```

```
proc return_confirm {confirm_url} {
    global new_account_link demo_details

set msg {[cgi_begin "Open Market Payment"]
You have selected an item that requires payment:
[payment_details]
If you have an Open Market account, click on "continue" below
and you will be prompted for your
account name and password.  If you do not have an account, you can
establish one on-line and return to this page to continue your
purchase.
<p><a href="$new_account_link">
<img src ="/images/open-button.gif"> an account on-line.</a><p>
<a href="$confirm_url"><img src="/images/continue-button.gif">
with payment transaction.</a>
$demo_details
[cgi_end]
}
    puts [subst $msg]
    exit
}

# ------------------------------------------------------------------------
#
#  Return a form rejecting the challenge response that was entered.
#
proc return-bad-response {} {
set msg {[cgi_begin "Open Market Bad Response"]
You have entered an invalid payment response.  Without the
correct response, we cannot process your payment transaction.
<p>Please return to the previous page (using your browser's
``Back'' function) and enter the correct information.
[cgi_end]
}
    puts [subst $msg]
    exit
}

set response_bad {[cgi_begin "Open Market Bad Response"]
You have entered an invalid response to a challenge.  Without the
correct response, we cannot process your payment transaction.
<p>Please return to the previous page (using your browser's
``Back'' function) and try again.
[cgi_end]
}

set response_retry_limit {[cgi_begin "Open Market Retry Limit"]
For security reasons, we only allow a certain number of tries
to get a correct response.  We are unable to process your transaction.
[cgi_end]
}

set payment_duplicate {[cgi_begin "Duplicate Payment Declined"]
We have received an HTTP request which would have the effect of
duplicating a payment.  The most frequent cause of this is
"backing up" with your browser and re-fetching a page by accident.<p>
Open Market systems detect this case and return this page instead
of allowing a duplicate payment.  (This is subtly different than
deliberately buying something twice - which is fine by us.)<p>
Thank you for using Open Market.
[cgi_end]
}
# ------------------------------------------------------------------------
#
#  Parse the URL:  extract the hash (signature) from the front of the URL,
```

```
#  and parse the remainder of the URL.
#

# this block of code sets up default values for important fields
set fields(url) ""
set fields(hid) ""
set fields(cc) "US"
set fields(amt) ""
set fields(hosh) ""
set fields(domain) ""
# default expiration is 30 days
set fields(expire) "2592000"
set fields(desc) "(unknown transaction)"
# last valid time
set fields(valid) 2147483647
set fields(fmt) "int"
set fields(billto) 0

# parse the URL, validate key and signature
cgi_parse_querystring $env(QUERY_STRING) fields secretkey merchant

if [info exists fields(mhid)] {
    set hid_split [split $fields(mhid) .]
    if {[llength $hid_split] != 2} {
        cgi_error verify_signature "hid_split bad format: $env(QUERY_STRING)"
    }
    set mid [lindex $hid_split 0]
    set hid [lindex $hid_split 1]
    if {[db_read_record secretkey secretkey_id $hid secretkey] != 1} {
        cgi_error mhid "bad key $mhid: $env(QUERY_STRING)"
    }

    if {$mid != $secretkey(principal_id) } {
        cgi_error mhid "merchant keyid mismatch: $env(QUERY_STRING)"
    }

    if {[db_read_record principal principal_id $mid merchant] != 1} {
        cgi_error mhid "bad merchant $mid: $env(QUERY_STRING)"
    }

}

# check for expiration of the payment URL
if {[currenttime] > $fields(valid)} {
    cgi_error validuntil "expired payment URL: $env(QUERY_STRING)"
}


# ---------------------------------------------------------------------
#
#  If the user ID wasn't supplied (i.e. it hasn't been cached by the
#  client) and the "proconf" field doesn't exist (or is expired), return
#  a pre-confirmation page.  This page lists the merchant, item, and amount,
#  and asks the user to continue with the payment transaction.
#
#  This mechanism exists so that users without CMI accounts have an opportunity
#  to jump off and establish one.
#
if {![info exists env(REMOTE_USER)]} {
    if {![info exists fields(proconf)] || ($fields(proconf) < [currenttime])} {
        set fields(proconf) [expr [currenttime]+600]
        return_confirm [cgi_self_link {} fields]
    } else {
        cgi_return_auth_required
    }
}

# ---------------------------------------------------------------------
#
#  Verify user stuff:
#       - user ID was supplied
#       - valid user ID
#       - valid user password
#
#  Any errors here are reflected back to the client with an "authorization
#  required" message, which will make the client prompt for username/password.
#
pay_getuser user

if {![info exists fields(dupok)]} {
    set msg {[cgi_begin "Open Market Payment"]
You have clicked on a link which requires payment:<br>
[payment_details]
But our records show that you have already purchased this item.
<p>It may be that you would like to buy the item again, or
it may have happened by accident.  For example, you may be attempting
to purchase a subscription which overlaps an earlier subscription.<p>
You can
<ul>
<li> <a href="$url">Go directly to the previous item</a>
<li> <a href="$purl">Go ahead and buy the item again</a>
</ul>
[cgi_end]
}
    set dupid [duplicate_check $user(principal_id) $merchant(principal_id) \
            $fields(domain)]
    if {$dupid != 0} {
        db_read_record transaction_log transaction_log_id $dupid old
        set surl [pay_build_smartlink $old(domain) $old(url) \
                [expr $old(transaction_date) + $old(expiration)] \
                $dupid secretkey $fields(fmt)]

        set fields(dupok) [uid_get 1 [expr [currenttime] + 600]]
        set purl [cgi_self_link {} fields]
        puts [subst $msg]
        exit
    }
} else {
    # dupok field, so validate it:
}

#
# fetch principal_account records for both buyer and merchant
# in order to get the limit fields
#
#
# get primary account for user and merchant
#
proc defaultaccount { pid a_ary} {
    upvar $a_ary ary
    set cmd "select * from principal_account \
            where principal_id = $pid \
            and flags = \"p\""
    return [db_query_read $cmd ary]
}

if {![defaultaccount $user(principal_id) useracount]} {
    cgi_error defaultaccount "No account for user $user(principal_id)"
}
```

3 nph-payment.cgi Sun Oct 9 14:03:10 1994

```
if {$fields(billto) != 0} {
    set cmd "select * from principal_account \
            where principal_id = $merchant(principal_id) \
            and account_id = $fields(billto)"
    set rc [db_query_read $cmd merchantaccount]
    if {$rc == 0} {
        cgi_error billto "bad account  $env(QUERY_STRING"
    }
} else {
    if {![defaultaccount $merchant(principal_id) merchantaccount]} {
        cgi_error defaultaccount \
                "No account for merchant $merchant(principal_id)"
    }
}

#
# useraccount is the principal_account field for the user
# merchantaccount is the principal_account field for the merchant

# ------------------------------------------------------------------------
#
#  Check transaction amount:
#       - positive, but not excessive
#       - doesn't exceed the merchant's floor limit
#
#
if {$fields(amt) < 0 || $fields(amt) > $merchantaccount(max_threshold)} {
    cgi_error merchantlimit "$env(QUERY_STRING)"
}

#
# check user limits
#
if {$fields(amt) > $useraccount(max_threshold)} {
    set msg {[cgi_begin "Account Limit Exceeded"]
Your account profile is set up so that the maximum amount of a single
transaction is $useraccount(max_threshold).  This transaction is for
$fields(amt), which is too much.  You can change your own account
profile by going to customer service.
[cgi_end]
}
    puts [subst $msg]
    exit
}

#
# challenge
#
if {![info exists fields(response)] && \
        ($fields(amt) >= $useraccount(confirm_threshold))} {
    # find out if the user has a scheme registered
    if {[db_query_read "select * from principal_authentication \
            where principal_id = $user(principal_id)" a]} {
        set scheme $a(scheme)
    } else {
        set scheme uc
    }
    set fname $library/auth-$scheme.tcl
    if {![file exists $fname]} {
        cgi_error challenge "File missing $fname"
    }
    source $fname
    auth_getchallenge user challenge
    log "$scheme challenge $challenge(challenge_name) response \
            $challenge(challenge_value)"
    set fields(response) \
            [md5 "[string tolower $challenge(challenge_value)] $secretkey(secret
    set fields(as) $scheme
#       set fields(postconf) [expr [currenttime]+600]
    set fields(uid) [uid_get 2 [expr [currenttime] + 600]]
    auth_return_challenge [cgi_self_link {} fields] $env(REMOTE_USER) \
            $challenge(challenge_name)
}



# ------------------------------------------------------------------------
#
#  If a response was requested, check response.  This should be a POST
#  method, and the hash of the reponse should match the expected field.
#  If they don't deny payment and return an error.
#
#  The response needs some "salt", instead of being a simple MD5 hash
#  of the expected result, to prevent dictionary attacks.
#
if {[info exists fields(as)] {
    set fname $library/auth-$fields(as).tcl
    if {![file exists $fname]} {
        cgi_error challenge "File missing $fname"
    }
    source $fname
    # now the authentication scheme procedures are defined
    if {$env(REQUEST_METHOD) == "POST"} {
        if {![info exists env(CONTENT_LENGTH)]} { set env(CONTENT_LENGTH) 0 }
        cgi_parse [read stdin $env(CONTENT_LENGTH)] postfields
    } else {
        # now check for a response in the query
        set query [split $env(QUERY_STRING) ?]
        if {[llength $query] > 1} {
            cgi_parse [lindex $query 1] postfields
        } else {
            cgi_error_list [list \
                    [list prog no_response] \
                    [list get  $env(QUERY_STRING)]]
        }
    }

    switch -- [auth_validateresponse fields postfields] {
        -2 {
            puts [subst $response_retry_limit]
            exit
        }
        -1 {
            puts [subst $response_bad]
            exit
        }
    }
    auth_nextchallenge $user(principal_id)
}

#
# last backstop, check the uid, if there is one, to reject bad duplicates
#
if {[info exists fields(dupok)] {
    if {[uid_valid $fields(dupok)] == 0} {
        # this is a duplicate, and probably innocuous, so just return
        # a friendly page, but log the event, for now
        puts [subst $payment_duplicate]
        syslog_log [list \
                [list prog uid_dup] \
```

```
                [list get $env(QUERY_STRING)]]
        exit
    }
}


#
#  Enter the transaction into the transaction database
#
set date [currenttime]
set tid [enter_transaction $user(principal_id) $merchant(principal_id) \
    $useraccount(account_id) $merchantaccount(account_id) \
    $fields(amt) $fields(cc) $env(REMOTE_ADDR) $fields(domain) \
    $fields(expire) \
    $fields(url) $fields(desc) $date $fields(fmt)]

# ------------------------------------ ----------------------------------------
#
#  Write a URL that grants access to this domain for a client
#  coming from the specified IP address.  Return a redirect that reflects
#  the client to the real URL. Put the transaction id into the
#  access URL in case we want to understand its origin
#

set aurl(domain) $fields(domain)
set aurl(expire)  [expr $date + $fields(expire)]
set aurl(ip) $env(REMOTE_ADDR)
set aurl(tid) $tid

set url [pay_accessurl $fields(url) aurl secretkey $fields(fmt)]

puts [cgi_return_redirect $url "Payment Accepted" \
    "Your payment has been accepted and processed." \
    "Select here to continue."]

}
# end of run_with_log

# XXX need to use payment server keys for intermediate steps
```

```
#!/usr/local/bin/tclsh-misc
#
#  Part of payment system prototype
#
#   This is the CGI script that takes a customer name
#   and produces a smart statement
#
#  Lawrence C. Stewart
#  stewart@openmarket.com
#


#
#  Load support routines.
#
set library {../lib}
source $library/mail.conf
source $library/log.tcl
run_with_log {

source $library/database.tcl
source $library/ticket.tcl
source $library/cgilib.tcl
source $library/html.tcl
source $library/payment.tcl
source $library/smartstatement.tcl

syslogInit nph-statement.cgi pid

open_database_as_statement

#  Verify user identity

pay_getuser user

# OK, now we have a valid account, so build the statement


# handle processing of commands
if {![info exists env(QUERY_STRING)]} { set env(QUERY_STRING) ""}

# parse the GET fields, if any, into fields
if {[string length $env(QUERY_STRING)] == 0} {
    smart_statement user
} else {
    cgi_parse_querystring $env(QUERY_STRING) fields omi omisecretkey

    if {![cgi_fieldcheck fields id]} {
        cgi_error GET_missing_id [array_to_list fields]
    }

    if {$user(principal_id) != $fields(id)} {
        cgi_error user_mismatch [concat [array_to_list fields] \
                [list user $user(principal_id)]]
    }
    if {![cgi_fieldcheck fields op]} {
        cgi_error missing_op [array_to_list fields]
    }

    switch $fields(op) {
        interval {
            if {![cgi_fieldcheck fields {first last}]} {
                cgi_error op_remove "missing first or last"
            }
            smart_statement user $fields(first) $fields(last)
            exit
        }
    }
}
}
}
```

```
#!/usr/local/bin/tclsh-misc
#
#  Part of payment system prototype
#
#   This is the CGI script that takes a customer name
#   and produces a shopping cart statement
#
#  Lawrence C. Stewart
#  stewart@openmarket.com
#


#
#  Load support routines.
#
set library {../lib}
source $library/mall.conf
source $library/log.tcl
run_with_log {

source $library/database.tcl
source $library/ticket.tcl
source $library/cgilib.tcl
source $library/html.tcl
source $library/payment.tcl
source $library/shoppingcart.tcl


set returnpage {[cgi_begin "Operation Completed"]
[cgi_link $env(SCRIPT_NAME) "Return to Shopping Cart"]
[cgi_end]
}

set purchasemsg {[cgi_begin "Purchase"]
This function not implemented yet.<p>
What will happen is that the items from a particular merchant
will be purchased as a unit
and represented as a single item on your smart statement.  The
statement in turn will link back to descriptions of the
individual items.
<p>[cgi_link $env(SCRIPT_NAME) "Return to Shopping Cart"]
[cgi_end]
}

syslogimit nph-cart.cgi pid

open_database

#  Verify user identity

pay_getuser user

# handle processing of commands
if {![info exists env(QUERY_STRING)]} { set env(QUERY_STRING) ""}

# parse the GET fields, if any, into fields
if {[string length $env(QUERY_STRING)] == 0} {
    # there is no query yet, so this must be the initial entry
    # into the system.
    #
    # get necessary information to write tickets
    #
    db_read_record principal access_name \"openmarket@openmarket.com\" omi
    pay_prin_to_key omi omisecretkey

    # return a screen with initial choices]
    send_cart_view user

} else {
    #
    # process GET string, making sure that it has not been tampered with
    #
    cgi_parse_querystring $env(QUERY_STRING) fields omi omisecretkey

    if {![cgi_fieldcheck fields id]} {
        cgi_error GET_missing_id [array_to_list fields]
    }

    if {$user(principal_id) != $fields(id)} {
        cgi_error user_mismatch [concat [array_to_list fields] \
                [list user $user(principal_id)]]
    }
    if {![cgi_fieldcheck fields op]} {
    }

    switch $fields(op) {
        remove {
            if {![cgi_fieldcheck fields cid]} {
                cgi_error op_remove "missing cid"
            }
            db_delete_row scart_item $fields(cid)
            puts [subst $returnpage]
            exit
        }
        removecart {
            if {![cgi_fieldcheck fields cid]} {
                cgi_error op_removecart "missing cid"
            }
            execsql "delete from scart_item \
                    where shoppingcart_id = $fields(cid)"
            db_delete_row shoppingcart $fields(cid)
            puts [subst $returnpage]
            exit
        }
        purchase {
            if {![cgi_fieldcheck fields cid]} {
                cgi_error op_purchase "missing cid"
            }
            puts [subst $purchasemsg]
            exit
        }
    }
}


}
```

```
#!/usr/local/bin/tclsh-misc
#
#   This is the CGI script that permits a buyer to add an item
#   to a shopping cart.
#
#   L. Stewart
#   stewart@openmarket.com
#


#
#  Load support routines.
#
set library {../lib}
source $library/mall.conf
source $library/log.tcl
source $library/database.tcl
source $library/ticket.tcl
source $library/cgilib.tcl
source $library/payment.tcl

# handle errors
run_with_log {

set toomany {[cgi_begin "Too many items"]
We're sorry, shopping carts can only hold 25 items,
so we are unable to add one more.<p>
You can use the "back up" feature of your browser
to return to the page you were looking at before.
[cgi_end]
}

#
# new_account_link are imported from mall.conf
#
proc return_confirm {confirm_url} {
    global new_account_link

set msg {[cgi_begin "Open Market Shopping Cart"]
You have selected an item to be added to your shopping cart.<p>
If you have an Open Market account, click on "continue" below
and you will be prompted for your
account name and password.  If you do not have an account, you can
establish one on-line and return to this page to continue.

<p><a href="$new_account_link">
<img src ="/images/open-button.gif"> an account on-line.</a><p>
<a href="$confirm_url"><img src="/images/continue-button.gif">
with your transaction.</a>
[cgi_end]
}
    puts [subst $msg]
    exit
}


# define default logging headers
syslogInit nph-c1.cgi pid

# open database in read-write mode
open_database

#
# process the URL, to see if it is OK
#
```

```
# this block of code sets up default values for important fields
set fields(url) ""
set fields(aurl) ""
set fields(kid) ""
set fields(cc) "US"
set fields(amt) ""
set fields(domain) ""
# default expiration is 30 days
set fields(expire) "2592000"
set fields(desc) "(unknown transaction)"
# last valid time
set fields(valid) 2147483647
set fields(billto) 0
set fields(detail) ""
set fields(qty) 1
#
# XXX what about fmt?
#

# parse the URL, validate key and signature
cgi_parse_querystring $env(QUERY_STRING) fields secretkey merchant

# check for expiration of the payment URL
if {[[currenttime] - $fields(valid)] > 0} {
    cgi_error validuntil "expired payment URL: $env(QUERY_STRING)"
}

#
#find out who is the user
#

#  If the user ID wasn't supplied (i.e. it hasn't been cached by the
#  client) and the "preconf" field doesn't exist (or is expired), return
#  a pre-confirmation page.
#
#  This mechanism exists so that users without ONE accounts have an opportunity
#  to jump off and establish one.
#
if {![info exists env(REMOTE_USER)]} {
    if {![info exists fields(preconf)] || ($fields(preconf) < [currenttime])} {
        set fields(preconf) [expr [currenttime]+600]
        return_confirm [cgi_self_link {} fields]
    } else {
        cgi_return_auth_required
    }
}

# ------------------------------------------------------------------------
#
#  Verify user stuff:
#       - user ID was supplied
#       - valid user ID
#       - valid user password
#
#  Any errors here are reflected back to the client with an "authorization
#  required" message, which will make the client prompt for username/password.
#
pay_getuser user

#
# find out if there is an existing shopping cart
# if not, create one
#

set query "select * from shoppingcart \
```

```
        where principal_id = $user(principal_id) \
        and merchant_id = $merchant(principal_id) \
        and purchased = 0 \
        and (expiration_date - [currenttime]) > 0"

if {![db_query_read $query cart]} {
    set cart(principal_id) $user(principal_id)
    set cart(merchant_id) $merchant(principal_id)
    set cart(create_date) [currenttime]
    set cart(expiration_date) [expr [currenttime] + 86400]
    set cart(purchased) 0
    db_insert_row shoppingcart cart
}

#
# at this point, there is a shopping cart, maybe with nothing in it
#

#
# The new item is always added to the cart.
#
# XXX denial of service attack!  What happends on add-to-cart at high speed?
# need to limit quantity of items in a cart
#

execsql "select count(*) from scart_item \
        where shoppingcart_id = $cart(shoppingcart_id)"
set count [sybnext $syb]

if {$count >= 25} {
    puts [subst $toomany]
    exit
}

set item(shoppingcart_id)  $cart(shoppingcart_id)
set item(amount) $fields(amt)
set item(currency) $fields(cc)
set item(transaction_date) [currenttime]
set item(domain) $fields(domain)
set item(expiration) $fields(expire)
set item(url) $fields(url)
set item(murl) $fields(murl)
set item(description) $fields(desc)
set item(valid_until) $fields(valid)
set item(detail) $fields(detail)
set item(quantity) $fields(qty)
db_insert_row scart_item item

#
# return a view of the cart itself
#
source $library/shoppingcart.tcl

send_cart_view user $item(shoppingcart_id) $item(scart_item_id)

}
# end of run_with_log
```

3 nph-c1.cgi Wed Oct 12 16:10:29 1994

```
#!/usr/local/bin/tclsh-misc
#
#  Part of payment system prototype
#
#    This is the CGI script that takes an access URL
# for a shopping cart.
#  It marks the cart paid, if it isn't already, and causes
#  the order to be sent to the vendor.
#
#  Lawrence C. Stewart
#  stewart@openmarket.com
#

#
#  Load support routines.
#
set library {../lib}
source $library/mall.conf
source $library/log.tcl
run_with_log {

source $library/database.tcl
source $library/ticket.tcl
source $library/cgilib.tcl
source $library/html.tcl
source $library/payment.tcl
source $library/shoppingcart.tcl


set returnpage {[cgi_begin "Operation Completed"]
[cgi_link $env(SCRIPT_NAME) "Return to Shopping Cart"]
[cgi_end]
}

set purchasemsg {[cgi_begin "Purchase"]
This function not implemented yet.<p>
What will happen is that the items from a particular merchant
will be purchased as a unit
and represented as a single item on your smart statement.  The
statement in turn will link back to descriptions of the
individual items.
<p>[cgi_link $env(SCRIPT_NAME) "Return to Shopping Cart"]
[cgi_end]
}

syslogint nph-cartaccess.cgi pid

open_database

#  Verify user identity

pay_getuser user

# handle processing of commands
if {![info exists env(QUERY_STRING)]} { set env(QUERY_STRING) ""}

# parse the GET fields, if any, into fields
if {[string length $env(QUERY_STRING)] == 0} {
    # there is no query yet, so this must be the initial entry
    # into the system.
    #
    # get necessary information to write tickets
    #
    db_read_record principal access_name \"openmarket@openmarket.com\" omi
    pay_priv_to_key omi omisecretkey

    # return a screen with initial choices)

    send_cart_view user

} else {
    #
    # process GET string, making sure that it has not been tampered with
    #
    cgi_parse_querystring $env(QUERY_STRING) fields omi omisecretkey


    if {![cgi_fieldcheck fields {domain expire tid}]} {
        cgi_error GET_missing_id [array_to_list fields]
    }

    # now convert domain into cart id
    regexp {cart-(.*)} $fields(domain) xx cartid
    if {![info exists cartid]} {
        cgi_error bad_cartid [array_to_list fields]
    }
    if {![db_read_row shoppingcart $cartid old]} {
        cgi_error bad_cart [array_to_list fields]
    }
    if {$old(purchased) == 0} {
        set new(purchased) 1
        set rc [db_update_row shoppingcart $cartid old new]
        switch -- $rc {
            -3 { cgi_error bad_cart }
            -1 { cgi_error match_error }
        }
        # cart is now marked purchased, and we now need to send the order

    }
    send_invoice_view user $cartid $fields(tid)
}


}
```

```
#
#  Common routines for talking to the payment database
#
# L. Stewart
# stewart@openmarket.com

# library has to be defined
source $library/dbschema.tcl

# also requires
# source $library/log.tcl

#
# ------------------------------------------------------------------------
#
#  Open up the payment system database; leave database handle in $syb
#  use this one for read-write purposes
#
proc open_database {} {
    global syb rwaccount rwpassword
    set syb [sybconnect $rwaccount $rwpassword]
}
#
# ------------------------------------------------------------------------
#
#  Open up the payment system database; leave database handle in $syb
#  use this one for getting statements or other read-only info
#
proc open_database_as_statement {} {
    global syb roaccount ropassword
    set syb [sybconnect $roaccount $ropassword]
}


# ------------------------------------------------------------------------
#
#  Run a SQL command, raising a Tcl error (with the SQL error text) if
#  it completed with error
#
# XXX what does the error call do?
#
proc execsql {cmd} {
    global syb sybmsg

    if {catch {sybsql $syb $cmd} msg} {
        error "execsql error <$msg> in $cmd: $sybmsg(msgtext)"
    }
    return $msg
}

# ------------------------------------------------------------------------
#
#  Load a (row) result into Tcl array
#
proc loadresult {array} {
    global syb sybmsg
    upvar $array a
    set row [sybnext $syb]
    set nextrow $sybmsg(nextrow)
    foreach column [sybcols $syb] {
        set a($column) [lindex $row 0]
        set row [lrange $row 1 end]
    }
   set sybmsg(nextrow) $nextrow
}
```

```
#
# read record
#
# select the record in <table> where field = value
# and load all the fields into ary
#
# if the value needs to be quoted in the SQL query, that has to be done
# by the caller
#
proc db_read_record {table field value ary} {
  upvar $ary a
  global syb
  if [info exists a] {unset a}
  case [sybsql $syb "select * from $table where $field = $value"] {
       {NO_MORE_ROWS} {
            return 0
       }
       {REG_ROW} {
           loadresult a
           return 1
       }
    }
}

#
# read a record given a sql query
#
proc db_query_read {query a_ary} {
    upvar $a_ary ary
    global syb
    if [info exists ary] { unset ary }
    case [sybsql $syb $query] {
        {NO_MORE_ROWS} {
            return 0
        }
        {REG_ROW} {
            loadresult ary
            return 1
        }
    }
}


#
#  Get the principal record for name and return it in array
#
proc get_principal {name array} {
    upvar $array a
    return [db_read_record principal access_name "\"$name\"" a]
}

proc insert_row {table a_ary} {
    upvar $a_ary p
    db_insert_row $table p
}

proc db_insert_row {table valuearray} {
    global syb tables identity
    upvar $valuearray p
    set id ""
    set elements $tables($table)
    set query "insert $table ([join [fieldnames_noid $table] ","])
               values ( [subst $elements] )"
```

1 database.tcl Mon Oct 10 13:57:59 1994

```
    execsql $query
    sybsql $syb "select @@identity"
    set id [sybnext $syb]
    set p($identity($table)) $id
    syslog_log [list [list prog db_insert_row] [list table $table] \
            [array_to_list p]]
    return $id
}



# useful routines
#

proc fieldnames {table} {
    global tables identity
    if {![info exists tables($table)]} {
        set res ""
    } else {
        regsub -all {([\$p\(]|\))} $tables($table) "" x2
        regsub -all {([\\\"])} $x2 "" x3
        regsub -all "\[ \t\r\n\]" $x3 "" x4
        set res [split $x4 ,]
    }
    if {info exists identity($table)} {lappend res $identity($table)}
    set res
}


proc fieldnames_noid {table} {
    global tables identity
    if {![info exists tables($table)]} {
        set res ""
    } else {
        regsub -all {([\$p\(]|\))} $tables($table) "" x2
        regsub -all {([\\\"])} $x2 "" x3
        regsub -all "\[ \t\r\n\]" $x3 "" x4
        set res [split $x4 ,]
    }
    set res
}



proc quotedfields {table ary} {
    global tables identity
    upvar $ary a
    if {info exists a} { unset a}
    regsub -all {([\$p\(]|\))} $tables($table) "" x2
    foreach el [split $x2 ,] {
        set n [string trim $el]
        set a([string trim $n {\"}]) [regexp  {\\\"[^\"]+\\\"}  $n]
    }
    if {info exists identity($table)} {set a($identity($table)) 0}
}

# the array new has new(field) = value pairs.  Each one is
# checked against the requirements put upon it by the database
# Fields that need to be quoted are quoted.
# If any fields fail the checks, then the procedure returns 0
#
proc db_validatefields { table a_new } {
    upvar $a_new new

    # field validation goes here

    # 1. trim whitespace and quotes
    # 2. check for bad characters
    # 3. check for size limits
    foreach item [array names new] {
        regsub -all {"} $new($item) { } t
        set new($item) [string trim $t]
    }

    # quote all fields in new which need it
    quotedfields $table qf
    foreach item [array names new] {
        if {$qf($item)} { set new($item) \"$new($item)\" }
    }
}

#
# db_update_row table id old new
#
# updates one or more fields in a row.
# table specifies which table
# id    selects a record (this is a value in the $identity(table) column
# old and new are arrays containing the old and new values for the fields
# with old(field) == value
# a transaction is opened, the record is read, if all the old values
# match their current values, then the values in new are substituted.
# There is no need for the items in old to be the same as those in new
#
# If a match failure occurs, a syslog entry is written and the procedure
# returns -1
# If the update fails, then the procedure returns -2
#
proc db_update_row {table id a_match a_new} {
    global identity
    upvar $a_match match
    upvar $a_new new
    if {![db_read_record $table $identity($table) $id current]} {
        syslog_log [list [list prog db_update_row-missing] \
                [list table $table] [list id $id]]
        return -3
    }
    foreach item [array names match] {
        # XXX bug here, numerics read as .0
        if {[string compare [string tolower $match($item)] \
                [string tolower $current($item)]] != 0} {
            syslog_log [list [list prog db_update_row-mismatch] \
                    [list table $table] \
                    [list current [array_to_list current]] \
                    [list match [array_to_list match]] \
                    ]
            return -1
        }
    }
    # validate new record fields
    db_validatefields $table new

    set q "update $table set \n"
    foreach item [array names new] {
        lappend lr "$item = $new($item)"
        # record old state of items to be changed
        set old($item) $current($item)
    }
    append q [join $lr ",\n"]
    append q "\n where $identity($table) = $id\n"

    # we write the log record before doing the update because if
```

```
    # execsql fails, it will write a stack trace and exit
    # XXX really want to add transaction and do this with no abnormal
    # flow of control
    syslog_log [list [list prog db_update_row] [list table $table] \
            [list id $id] \
            [list old [array_to_list old]] \
            [list new [array_to_list new]] \
            ]
    # do the update
    # XXX check return!
    execsql $q

    return 0
}


#
# db_delete_row table id match
#
# deletes a row
#  table specifies which table
#  id    selects a record (this is a value in the $identity(table) column
#  match is an array containing values which have to match the current
#     row
#
#  a transaction is opened, the record is read, if all the match values
#  match their current values, then the row is deleted
#
# if the record doesn't exist, then the procedure returns -3
# if a match failue occurs, the procedure returns -1
# if the delete fails, then the procedure returns -2
# The procedure returns 0 on success
#
proc db_delete_row { table id {a_match xxx}} {
    global identity
    upvar $a_match match
    # the following two statemnts create an empty array
    # in the case that match was not passed in
    set match(xyzzy) plugh
    unset match(xyzzy)
    if {![db_read_record $table $identity($table) $id current]} {
        syslog_log [list [list prog db_delete_row-missing] \
                [list table $table] [list id $id]]
        return -3
    }
    foreach item [array names match] {
        # XXX bug here, numerics read as .0
        if {[string compare [string tolower $match($item)] \
                [string tolower $current($item)]] != 0} {
            syslog_log [list [list prog db_delete_row-mismatch] \
                    [list table $table] \
                    [list current [array_to_list current]] \
                    [list match [array_to_list match]] \
                    ]
            return -1
        }
    }

    set q "delete from $table where $identity($table) = $id"

    # we write the log record before doing the delete because if
    # execsql fails, it will write a stack trace and exit
    # XXX really want to add transaction and do this with no abnormal
    # flow of control
    syslog_log [list [list prog db_delete_row] [list table $table] \
            [list old [array_to_list current]] \
            ]
    # do the update
    # XXX check return!
    execsql $q

    return 0
}

#
# reads a record from the given table, matching on the identity column
#
proc db_read_row {table id ary} {
  upvar $ary a
  global syb identity
  if [info exists a] {unset a}
  case [sybeql $syb "select * from $table where $identity($table) = $id"] {
        {NO_MORE_ROWS} {
            return 0
        }
        {REG_ROW} {
            loadresult a
            return 1
        }
    ;
}


#
# map_query - call a procedure with rows matching a query
#
proc db_map_query {query pr} {
    global sybmsg
    set result [execsql $query]
    while {$sybmsg(nextrow) == "REG_ROW"} {
        loadresult temp
        if {$sybmsg(nextrow) != "REG_ROW"} { break }
        $pr temp
    }
}



#
# unique id system
#

#
# uid_get <times_valid> <expiration>
#
# uid_get will return an id that can be used exactly so many times
# In other words, it will "expire" either when it has been used so
# many times or when it reaches its expiration date
#
proc uid_get {{ times_valid  1} { expiration_date 2147483647} } {
    global syb
    # This service will bypass logging for insert_row
    execsql "insert ntimes values ( 0, $times_valid, $expiration_date )"
    sybeql $syb "select @@identity"
    set id [sybnext $syb]
    return $id
}

#
#
```

```
proc uid_valid {id} {
    if {![db_read_record ntimes ntimes_id $id a]} { return 0 }
    set now [currenttime]
    if {[[currenttime] - $a(expiration)] > 0} { return 0 }
    incr a(uses)
    if {$a(uses) > $a(maxuses)} { return 0 }
    execsql "update ntimes set uses = $a(uses) where ntimes_id = $id"
    return $a(uses)
}
```

7 database.tcl Mon Oct 10 13:57:59 1994

```
#
# Tables in pay941010
#
# This file generated automatically by getschema.tcl
# on Mon Oct 10 15:22:15 EDT 1994
#

# name of the payment database
set paydb pay941010

# sybase account used for read-write access to the database
set rwaccount pay941010
set rwpassword pay941010

# sybase account used for read-only access to the database
set roaccount state941010
set ropassword state941010


#
# Fields in pay941010.principal
#


# has identity field principal_id
set tables(principal) {
    \"$p(access_name)\",
    \"$p(access_password)\",
    \"$p(principal_name)\",
    \"$p(address_1)\",
    \"$p(address_2)\",
    \"$p(address_3)\",
    \"$p(postal_code)\",
    \"$p(country)\",
    \"$p(telephone)\",
    \"$p(fax)\",
    \"$p(email)\",
    \"$p(secret_key)\",
    $p(secretkey_id),
    \"$p(home_url)\",
    \"$p(status)\",
    \"$p(rtype)\"
}
set identity(principal) principal_id


#
# Fields in pay941010.secretkey
#


# has identity field secretkey_id
set tables(secretkey) {
    $p(principal_id),
    $p(expiration_date),
    \"$p(secret_key)\"
}
set identity(secretkey) secretkey_id


#
# Fields in pay941010.account
#


# has identity field account_id
set tables(account) {
    \"$p(card_number)\",
    \"$p(expiration_date)\",
    \"$p(billing_name)\",
    \"$p(address_1)\",
    \"$p(address_2)\",
    \"$p(address_3)\",
    \"$p(postal_code)\",
    \"$p(country)\",
    \"$p(currency)\",
    \"$p(status)\",
    \"$p(rtype)\"
}
set identity(account) account_id


#
# Fields in pay941010.balance
#


# has identity field balance_id
set tables(balance) {
    $p(account_id),
    $p(cash_balance)
}
set identity(balance) balance_id


#
# Fields in pay941010.principal_account
#


# has identity field principal_account_id
set tables(principal_account) {
    $p(principal_id),
    $p(account_id),
    $p(confirm_threshold),
    $p(max_threshold),
    \"$p(name)\",
    \"$p(flags)\",
    \"$p(status)\",
    $p(cmi_confirm),
    $p(cmi_max)
}
set identity(principal_account) principal_account_id


#
# Fields in pay941010.challenge
#


# has identity field challenge_id
set tables(challenge) {
    $p(principal_id),
    \"$p(challenge_name)\",
    \"$p(challenge_value)\"
}
set identity(challenge) challenge_id
```

```
#
# Fields in pay941010.transaction_log
#


# has identity field transaction_log_id
set tables(transaction_log) {
    $p(amount),
    \"$p(currency)\",
    $p(transaction_date),
    $p(initiator),
    $p(benificiary),
    $p(from_account),
    $p(to_account),
    \"$p(transaction_type)\",
    \"$p(ip_address)\",
    \"$p(domain)\",
    $p(expiration),
    \"$p(url)\",
    \"$p(description)\"
}
set identity(transaction_log) transaction_log_id


#
# Fields in pay941010.duplicate
#


# has identity field duplicate_id
set tables(duplicate) {
    $p(transaction_log_id),
    $p(initiator),
    $p(benificiary),
    \"$p(domain)\",
    $p(expiration)
}
set identity(duplicate) duplicate_id


#
# Fields in pay941010.authorize
#


# has identity field authorize_id
set tables(authorize) {
    $p(amount),
    \"$p(currency)\",
    $p(transaction_log_id),
    \"$p(status)\",
    \"$p(result)\",
    \"$p(request_type)\",
    $p(authorize_date)
}
set identity(authorize) authorize_id


#
# Fields in pay941010.nextchallenge
#


# has identity field nextchallenge_id
set tables(nextchallenge) {
    $p(principal_id),
    $p(challenge_id)
}
set identity(nextchallenge) nextchallenge_id


#
# Fields in pay941010.principal_authentication
#


# has identity field principal_authentication_id
set tables(principal_authentication) {
    $p(principal_id),
    \"$p(scheme)\"
}
set identity(principal_authentication) principal_authentication_id


#
# Fields in pay941010.softnetkey
#


# has identity field softnetkey_id
set tables(softnetkey) {
    $p(principal_id),
    \"$p(secret_key)\"
}
set identity(softnetkey) softnetkey_id


#
# Fields in pay941010.snk
#


# has identity field snk_id
set tables(snk) {
    $p(principal_id),
    \"$p(secret_key)\"
}
set identity(snk) snk_id


#
# Fields in pay941010.ntimes
#


# has identity field ntimes_id
set tables(ntimes) {
    $p(uses),
    $p(maxuses),
    $p(expiration)
}
set identity(ntimes) ntimes_id


#
# Fields in pay941010.shoppingcart
#


# has identity field shoppingcart_id
```

```
set tables(shoppingcart) {
    $p(principal_id),
    $p(merchant_id),
    $p(create_date),
    $p(expiration_date),
    $p(purchased)
}
set identity(shoppingcart) shoppingcart_id


#
# Fields in pay941010.scart_item
#


# has identity field scart_item_id
set tables(scart_item) {
    $p(shoppingcart_id),
    $p(amount),
    \"$p(currency)\",
    $p(transaction_date),
    \"$p(domain)\",
    $p(expiration),
    \"$p(url)\",
    \"$p(murl)\",
    \"$p(description)\",
    $p(valid_until),
    \"$p(detail)\",
    $p(quantity)
}
set identity(scart_item) scart_item_id


#
# Fields in pay941010.settle
#


# has identity field settle_id
set tables(settle) {
    $p(transaction_log_id),
    $p(settle_log_id)
}
set identity(settle) settle_id
```

5 dbschema.tcl Mon Oct 10 16:49:33 1994

```
#
# This file contains subroutines for shopping cart management
#

proc cart_make_payurl {total nitems currency} {
    global payment_server_root cart secretkey
    set paylinkbase "$payment_server_root/bin/nph-payment.cgi?"
    set fields(url) $payment_server_root/bin/nph-cartaccess.cgi
    set fields(amt) $total
    set fields(cc) $currency
    regsub "\.0$" $cart(shoppingcart_id) "" cartid
    set fields(domain) cart-$cartid
    set fields(desc) "$nitems items"
    set fields(fmt) get

    set paylink [sec_create_ticket secretkey fields]
    return $paylinkbase$paylink
}

proc my_self_link { l } {
  global env
  set old $env(SCRIPT_NAME)
  set env(SCRIPT_NAME) /bin/nph-cart.cgi
  set result [cgi_self_link $l]
  set  env(SCRIPT_NAME) $old
  return $result
}

proc send_cart_item { a_ci } {
    upvar $a_ci ci
    global secretkey user
    global cart_total cart_items cart_currency
    incr cart_items
    if {[string compare $cart_currency ""] == 0} {
        set cart_currency $ci(currency)
    } else {
        if {[string compare $cart_currency $ci(currency)] != 0} {
            set cart_error "Cart contains multiple currencies!"
        }
    }
    set cart_total [expr $cart_total + $ci(amount)]
    set url [pay_build_smartlink $ci(domain) $ci(url) \
            [expr [currenttime] + 86400] \
            $ci(scart_item_id) secretkey int]
    set purchase [my_self_link [list \
            [list op purchase] \
            [list cid $ci(scart_item_id)] \
            [list id $user(principal_id)] \
            ]]
    set remove [my_self_link [list \
            [list op remove] \
            [list cid $ci(scart_item_id)] \
            [list id $user(principal_id)] \
            ]]
    puts "<br><a href=\"$purchase\"> \
                <img src=\"../images/buy-item-button.gif\" \
                alt=\"\[ Buy Item \]\"></a> \
                <a href=\"$remove\"> \
                <img src=\"../images/delete-item-button.gif\" \
                alt=\"\[ Delete Item \]\"></a> \
                <a href=\"$url\"> $ci(description) ... $ci(amount) \
            ($ci(currency))</a>"
}
```

```
proc send_invoice_item { a_ci } {
    upvar $a_ci ci
    global secretkey user
    global cart_total cart_items cart_currency
    incr cart_items
    set cart_currency $ci(currency)

    set cart_total [expr $cart_total + $ci(amount)]
    if {[string length $ci(aurl)] > 0} {
        set url [pay_build_smartlink $ci(domain) $ci(aurl) \
                $ci(expiration) \
                $ci(scart_item_id) secretkey int]
    } else {
        set url [pay_build_smartlink $ci(domain) $ci(url) \
                [expr [currenttime] + 86400] \
                $ci(scart_item_id) secretkey int]
    }
    puts "<li> <a href=\"$url\">   $ci(description) ... $ci(amount) \
            ($ci(currency))</a>"
}


# send the user a view of the cart
# if shoppingcart_id is present, restrict the view to only
# the relevant manufacturer
#
# if scart_item_id is present, explain that this is the newest item
#
proc send_cart_view { a_user {shoppingcart_id 0} {scart_item_id 0}} {
    upvar $a_user user
    global secretkey sybmsg merchant cart
    global cart_total cart_items cart_currency

    # first build a list of the cart id's to represent
    if {$shoppingcart_id != 0} {
        lappend idlist $shoppingcart_id
    } else {
        set query "select * from shoppingcart \
                where principal_id = $user(principal_id) \
                and purchased = 0 \
                and (expiration_date - [currenttime]) > 0"
        set result [execsql "$query"]
        while {$sybmsg(nextrow) == "REG_ROW"} {
            loadresult ctemp
            if {$sybmsg(nextrow) != "REG_ROW"} { break }
            lappend idlist $ctemp(shoppingcart_id)
        }
    }

    puts [cgi_begin]
        puts "<TITLE>Shopping cart for $user(principal_name)</TITLE>"
        puts "<H1><IMG SRC=\"../images/omicon50.gif\" ALT=\"Open Market\">"
        puts "Shopping cart for $user(principal_name)</H1>"

    #
    puts {Your shopping cart can include items from one or more
    merchants.  Each item is a link that will take you back to
    the page on which that item is found.  In addition, each item
    has a "put back" button that will remove the item from the shopping
    cart.  Finally, you can purchase the items from each merchant.<p>
    A shopping cart will remain active for 24 hours.<p>}

    if {![info exists idlist]} {
        puts "Your shopping cart is empty."
        puts [cgi_end]
```

```
        exit
    }

    foreach cartid $idlist {
        if {![db_read_row shoppingcart $cartid cart]} {
            cgi_error send_cart_view "Cart $cartid gone missing"
        }

        if {![db_read_row principal $cart(merchant_id) merchant]} {
            cgi_error send_cart_view "Cart $cartid merchant gone missing"
        }
        pay_prin_to_key merchant secretkey
        set cart_total 0
        set cart_items 0
        set cart_currency ""
        puts "<h2>Items from $merchant(principal_name)</h2>"
        puts "These items will remain in the shopping cart until \
                [ctime $cart(expiration_date)]<p>"
        puts "<ul>"
        set query "select * from scart_item \
                where shoppingcart_id = $cartid"
        db_map_query $query send_cart_item
        puts "</ul>"

        puts "[cgi_link \
                [cart_make_payuri $cart_total $cart_items $cart_currency] \
                "<IMG SRC=\"../images/buy-all-button.gif\" ALT=\"Buy all items\"
                from $merchant(principal_name), \
                total is $cart_total ($cart_currency)"]<br>"
    }

        puts "[cgi_link [my_self_link \
                [list [list op removecart] [list cid $cartid] \
                [list id $user(principal_id)] \
                ] \
                ] "<IMG SRC=\"../images/empty-cart-button.gif\" ALT=\"Empty cart
                of all items from $merchant(principal_name)"]<br>"

    puts [cgi_end]
}


# send the user a view of the purchased collection of goods
# the arguments are the shopping cart id and tid
proc send_invoice_view { a_user shoppingcart_id tid} {
    upvar $a_user user
    global secretkey sybmsg merchant cart
    global cart_total cart_items cart_currency

    # first build a list of the cart id's to represent
    set cartid $shoppingcart_id


    if {![db_read_row shoppingcart $cartid cart]} {
        cgi_error send_invoice_view "Cart $cartid gone missing"
    }
    if {![db_read_row principal $cart(merchant_id) merchant]} {
        cgi_error send_invoice_view "Cart $cartid merchant gone missing"
    }
    if {![db_read_row transaction_log $tid trans]} {
        cgi_error send_invoice_view "Cart $cartid transaction gone missing"
    }
    pay_prin_to_key merchant secretkey
    set cart_total 0
    set cart_items 0
    set cart_currency ""
    puts [cgi_begin "Goods purchased from $merchant(principal_name)"]

    puts "<h2>Items</h2>"
    puts {These are the items included in this purchase.  Each item
    is a link that will take you back to the page from which the
    item was purchased.}
    puts "<ul>"
    set query "select * from scart_item \
            where shoppingcart_id = $cartid"
    db_map_query $query send_invoice_item
    puts "</ul>"


    puts [cgi_end]
}
```

```
#
#
# build smart statement for user
#
# L. Stewart
# stewart@openmarket.com
#



# parray is an array containing the principal record from the
# payment database

# the database should be open as global syb

proc smart_statement { a_prin {first 0} {last 0x7fffffff}} {
  upvar $a_prin user
  global sybmsg env payment_server_root secretkey

    # OK, now we have a valid account, so build the statement
    #

    puts [cgi_begin "Smart Statement for $user(principal_name)"]

    puts "Your Smart Statement is a record of recent transactions
you have made on the network. Each line contains the date of the
transaction, the merchant involved, a description of the item purchased,
and the amount. <p>
You will notice that the item description is in fact a hypertext link.
Where this link goes depends on what kind of item is involved:\n"

    puts "<ul>
<li>Hard Goods - The link goes to the current order status
<li> Information product - The link goes to the item you bought
<li> Information service - The link goes to more information about the item
</ul>
"

puts "In addition, the date field of an item is also a link.  This
link will take you to more detailed information about the item.<p>
"

if {$first == 0} {
    set dtl [fmtclock [currenttime] "%Y %m"]
    set first [convertclock "[lindex $dtl 1]/1/[lindex $dtl 0]"]
    set last [currenttime]
}
set my [fmtclock $first "%B %Y"]

puts "<h2>Transactions in [lindex $my 0], [lindex $my 1]</h2>"

# read all the transactions, and put them into an array

set query  "select * from transaction_log
   where (initiator = $user(principal_id)) and
   (transaction_date >= $first) and
   (transaction_date < $last)"

set result [execsql "$query"]

set numtrans 0
while {$sybmsg(nextrow) == "REG_ROW"} {
  loadresult res.$numtrans
  if {$sybmsg(nextrow) != "REG_ROW"} { break }
  incr numtrans
}

# go through the list, getting merchant information for
# all merchants who appear at least once

set i 0
while {$i < $numtrans} {
  set mvar res.${i}(beneficiary)
  set mid [set $mvar]
  if {![info exists merchant.${mid}(principal_id)] } {
      set ok [db_read_record principal principal_id $mid merchant.$mid]
      pay_prin_to_key merchant.${mid} secretkey.${mid}
  }
  incr i
}

db_read_record principal access_name \"openmarket@openmarket.com\" omi
pay_prin_to_key omi secretkey

# now process the whole list
set sum 0
set i 0
  puts "<P>"
while {$i < $numtrans} {
  set tvar res.${i}
  set mid [set ${tvar}(beneficiary)]
  if {[string index [set ${tvar}(transaction_type)] 0] == "g"} {
      set fmt get
  } else {
      set fmt int
  }
  set ourl [pay_build_smartlink  [set ${tvar}(domain)] \
           [set ${tvar}(url)] \
           [expr [set ${tvar}(transaction_date)] + [set ${tvar}(expiration)]] \
           [set ${tvar}(transaction_log_id)] secretkey.$mid $fmt]

  set tdate [string range [ctime [set ${tvar}(transaction_date)]] 0 9]
#
#
# create ticket link to detail server
#

  set nv(expire)  [expr [currenttime]+3600]
  set nv(id) [set ${tvar}(transaction_log_id)]
  set nv(ip) $env(REMOTE_ADDR)
  set nv(domain)  detail
  set domain statementdomain

  set detailurl [pay_accessurl $payment_server_root/bin/detail/detail.cgi \
        nv secretkey]

#

  puts "<a href=\"$detailurl\">$tdate</a>"
  puts " [set merchant.${mid}(principal_name)] "
  puts " <a href=\"$ourl\">[set ${tvar}(description)]</a>"
  puts "amount: \$ [set ${tvar}(amount)]"
  puts "<br>"
  set sum [expr $sum + [set ${tvar}(amount)]]
  incr i
  }


puts [format "<p>Your total is $%6.2f." $sum]
puts "<h1>Previous Statements</h1>"
```

1 smartstatement.tcl Sun Oct 9 17:25:19 1994

```
puts "<ul>"
set tk(op) interval
set tk(id) $user(principal_id)
set tk(last) $first
incr tk(last) -1
set my [fmtclock $tk(last) "%Y %m %e"]
set month [lindex $my 1]
set year [lindex $my 0]
set tk(first) [convertclock "$month/1/$year"]

set my [fmtclock $tk(first) "%B %Y"]

puts "<li> [cgi_link [cgi_self_link {} tk] "[lindex $my 0], [lindex $my 1]"]"

set tk(last) $tk(first)
incr tk(last) -1
set my [fmtclock $tk(last) "%Y %m %e"]
set month [lindex $my 1]
set year [lindex $my 0]
set tk(first) [convertclock "$month/1/$year"]

set my [fmtclock $tk(first) "%B %Y"]

puts "<li> [cgi_link [cgi_self_link {} tk] "[lindex $my 0], [lindex $my 1]"]"

puts "</ul>"
puts "Return to your [cgi_link $env(SCRIPT_NAME) "Newest Statement"]."

puts "<h2>Feedback</h2>"

puts "You can send us comments and suggestions "
puts [feedback-link here $smartStatement]
puts ".<p>"
end-html
exit

}
```

3 smartstatement.tcl Sun Oct 9 17:25:19 1994

```
#
#  General routines for manipulating URL name/value pairs and ticket
#  signatures.
#
#  Andrew Payne
#  payne@openmarket.com
#

# ------------------------------------------------------------------------
#
#  Return a string with "bad" characters escaped using the URL escaping
#  mechanism (i.e. "%XX" where "XX" is the hex representation for the
#  escaped character).
#
proc url-escape {what} {
    regsub -all {\%} $what "%25" what
    regsub -all {\@} $what "%23" what
    regsub -all {\/} $what "%2F" what
    regsub -all {\&} $what "%26" what
    regsub -all {\=} $what "%3D" what
    regsub -all { }  $what "%20" what
    regsub -all {\+} $what "%2B" what
    regsub -all {\:} $what "%3A" what
    regsub -all {\?} $what "%3F" what
    regsub -all {
} $what "%0A" what
    return $what
}

proc url_unparse {array} {
    upvar $array a
    foreach item [array names a] {
        lappend list "$item=[url-escape $a($item)]"
    }
    return [join $list "&"]
}

# ------------------------------------------------------------------------
#
#  Create a "ticket" of name/value pairs, signed by a specified hash.
#  The return string format is:
#
#       (hash):name1=value1&name2=value2...
#
proc create-ticket {key array} {
    upvar $array a

    set list {}
    foreach item [array names a] {
#       if {$a($item) != ""} {
# if != "" deleted by L. Stewart,  what was he thinking?
            lappend list "$item=[url-escape $a($item)]"
#       }
    }
    set string [join $list "&"]
    return "[md5 "$key $string"]:$string"
}


#
# wrapper for create-ticket which adds the keyid and calls
# create_ticket
#
proc sec_create_ticket {a_key array} {
    upvar $array a
    upvar $a_key key
    set a(kid) [pay_makekeyidpair $key(principal_id) $key(secretkey_id)]
    return [create-ticket $key(secret_key) a]
}


#
# sec_create_ticket_list is the same as sec_create_ticket
# except that nv is a name value list instead of an array
#
proc sec_create_ticket_list {a_secretkey list_nv} {
    upvar $a_secretkey secretkey
    list_to_array $list_nv nv
    return [sec_create_ticket secretkey nv]
}
```

1 ticket.tcl Mon Sep 26 13:08:25 1994

```
# why is this needed?  Don't we know what machine we are running on?

set payment_server_root "http://payment.openmarket.com"

# These items refer to the merchant server
#  Why is ther server name needed?  When is it used?

set merchant_server_root "http://www.openmarket.com"
set merchant_file_root  "/cmi/httpd/www/root"
set merchant_demo_path   "/demos/aug15"

#
#  This is the URL where the "open a new CMI account" points to.
#
set new_account_link $payment_server_root/service/establish.html

#
#  Details of the demo account.
#
set demo_details {
    <blockquote><p><i>NOTE:</i> For demonstrations, use the account name
    <b>testuser@openmarket.com</b> with the password <b>testuser</b>.
    </blockquote>
}
```

1 mall.conf Tue Sep 27 12:07:05 1994

```
# Tcl routines for logging from CGI scripts.
#
# Win Treese
# Open Market, Inc.
# treese@OpenMarket.com
#
# Created on Wed Jul 13 11:13:26 EDT 1994 by treese
# Last modified on Wed Jul 13 11:16:30 EDT 1994 by treese

proc run_with_log {script} {
    global argv0 errorInfo
    if [catch {uplevel #0 $script} errors] {
        set now [string trim [ctime [currenttime]]]
        puts stderr "\[$now\] $argv0: $errors"
        set dirs [split $argv0 /]
        set prog [lindex $dirs [expr [llength $dirs] - 1]]
        regsub -all "\n" $errorInfo "\n$prog: " backtrace
        puts stderr "\[$now\] $prog: $backtrace"
    }
}

# enter into the log
proc log {what} {
    global argv0
    set now [string trim [ctime [currenttime]]]
    puts stderr "\[$now\] $argv0: $what"
}


#
# syslog support
#

set syslog_facility local1
set syslog_level info

#
# syslog_init sets up the default string to be entered in syslog
# if pid == "pid" then the process id will be entered also
#
proc syslog_init {name pid} {
  syslogint $name $pid
}

#
# syslog_log enters the given text in the log using the default
# facility and level
#
proc syslog_log { t } {
  global syslog_facility syslog_level
  regsub -all {
} $t " " nt
  syslog $syslog_facility $syslog_level [concat [log_user] $nt]
# log " $syslog_facility $syslog_level [concat [log_user] $nt]"
}

proc log_user {} {
    global env user
    set uid 0
    set addr 0
    if [info exists env(REMOTE_ADDR)] { set addr $env(REMOTE_ADDR) }
    if [info exists user(principal_id)] { set uid $user(principal_id) }
    return [list [list uid $uid] [list addr $addr]]
}
```

```
#
# convert a tcl array (name value pairs) into a tcl list
# with elements being name-value 2 element lists
#
proc array_to_list {a_ary} {
  upvar $a_ary ary
  set res ""
  if [info exists ary] {
      foreach item [array names ary] {
          lappend res [list $item $ary($item)]
      }
  }
  return $res
}

#
# translate a list composed of name value pairs back into an array
#
proc list_to_array { l a_ary} {
  upvar $a_ary ary
  foreach item $l {
    set name [lindex $item 0]
    set value [lindex $item 1]
    set ary($name) $value
  }
}
```

1 log.tcl Tue Sep 6 13:46:35 1994

```
#
#  Library support routines for various HTTP returns.
#
#  Depends on:
#       html.tcl
#
#  Andrew Payne
#  payne@openmarket.com
#

# ------------------------------------------------------------------------
#
#  Return an "authorized required" reply to the client.  This will force
#  prompting for usern...e/password.
#
proc return-auth-required {{realm "Open Market Account"}} {
    log "return-auth-required $realm"
    puts "HTTP/1.0 401 Unauthorized"
    puts "Content-type: text/html"
    puts "WWW-Authenticate: Basic realm=\"$realm\""
    puts ""
    puts [title "Authorization Required"]
    puts "Browser not authentication-capable or authentication failed."
    puts "<p>"
    puts "The OpenMarket username and password you entered were not"
    puts "valid."
    end-html
    exit
}

# ------------------------------------------------------------------------
#
#  Return a redirect response to the client to the specified URL
#
proc return-redirect {url} {
    puts "HTTP/1.0 302 Found"
    puts "Location: $url"
    puts "Content-type: text/html"
    puts ""
    puts [title "Redirect"]
    puts "You appear to have a very old World Wide Web browser, that doesn't"
    puts "support the redirect operation.  We strongly suggest upgrading to"
    puts "the latest software.<p>"
    puts "Here's the <a href=\"$url\">actual document</a>."
    end-html
    exit
}

# ------------------------------------------------------------------------
#
#  Return an error page to the client.
#
proc return-error {} {
    puts "HTTP/1.0 200 OK"
    puts "Content-type: text/html"
    puts ""
    puts [title "Payment Transaction Error"]
    puts "An error occurred during the processing of your payment "
    puts "transaction. <p> The error has been logged to our attention."
    end-html
    exit
}
```

1 http.tcl Fri Aug 26 13:51:12 1994

```
#
#  Useful library routines for generating HTML
#
#  Andrew Payne
#  payne@openmarket.com
#

#
#  The merchant key and ID used for all files in this demo:
#
set merchantkey "testmerchant"
set merchantid  testmerchant@openmarket.com

# ---------------------------------------------------------------------
#
#  Shortcut routines for making HTML--these routines return their result
#
proc title {text} {
    return "<title>$text</title><h1>$text</h1>"
}

proc signature {} {
    global merchant_server_root merchant_demo_path
set msg {<p><hr><A HREF="$merchant_server_root/">
<IMG ALIGN="top" SRC="$merchant_server_root$merchant_demo_path/images/omicon32.g
<I>Copyright &#169; 1994 Open Market, Inc. All Rights Reserved.</I>
}
    return [subst $msg]
}

#
#  Generate a link to the feedback pages about the specified subject
#
proc feedback-link {text subject} {
    global payment_server_root
    set subject [url-escape $subject]
    return "<a href=\"$payment_server_root/bin/feedback.cgi?$subject\">$text</te
}

#
#  Generate a link to the dollar sign image
#
proc dollar-image {} {
    return "<img src=\"/images/dollar.gif\">"
}

#
#  Generate a link to the SmartStatement
#
proc online-statement-link {} {
    global payment_server_root

    return "<a href=\"$payment_server_root/bin/nph-statement.cgi\">
                    View your SmartStatement</a>"
}

# ---------------------------------------------------------------------
#
#  Shorthand for starting and ending HTML documents
#   (note that these routines write to stdout--they are intended to be
#   used in CGI scripts)
#
proc begin-html {title} {
    puts "Content-type: text/html"
    puts ""
    puts [title $title]
}

proc end-html {} {
    puts [signature]
}


# Create a user ticket

proc userurl {url id domain} {
    global env merchant_server_root merchant_demo_path
#    set nv(expire)  [expr [currenttime]+60000]
    set nv(domain) $domain
    set nv(principal) $id
    set secret_key testmerchant
    set hash [create-ticket "$secret_key $env(REMOTE_ADDR) $domain" nv]
    return "$merchant_server_root/0$hash$merchant_demo_path/members/$url"
}

# Write a hypertext link

set linklist ""
proc link {args} {
        global linklist
        parseargs argvals {-url -name} $args
        append linklist "<LI> <A HREF=\"$argvals(-url)\">$argvals(-name)</A>\n"
}

# Parse an argument vector.
# Usage: parseargs array-name list-of-options arg-vector
#   array-name is the name of an array that w'll get the options
#   list-of-options is a list of option names
#   arg-vector is the argument list to be processed.

proc parseargs {name argnames arglist} {
        upvar $name a
        foreach i $argnames {
                set a($i) ""
        }
        for {set i 0} {$i < [llength $arglist]} {incr i} {
                set opt [lindex $arglist $i]
                incr i
                set val [lindex $arglist $i]
                set a($opt) $val
        }
}

# Make hypertext links from a file.  Links are given on two lines:
# URL and then name.
# XXX needs error checking

#proc makelinks {filename} {
#       set f [open $filename r]
#       set links ""
#       while {[gets $f url] >= 0} {
#               gets $f name
#               append links "<LI> <A HREF=\"$url\">$name</A>\n"
#       }
#       return $links
#}

proc makelinks {filename} {
}
```

```
proc load {file} {
        global linklist
        source $file
        set retval $linklist
        set linklist ""
        return $retval
}
```

3 html.tcl Mon Oct 10 16:39:24 1994

```
#
#  Generally useful
#
#  L. Stewart
#  stewart@openmarket.com
#


# send a screen back to the user explaining something about the error
# and record the message in syslog
proc cgi_error { {branch ""} {etext ""} {opt ""}} {
    set msg {[cgi_begin "Transaction Error"]
An error has occurred during the processing of your transaction.
It has been logged to our attention.<p>
$opt
[cgi_end]
}
    puts [subst $msg]
    syslog_log [list [list prog $branch] [list text $etext]]
    exit
}

proc cgi_error_list { log_list {opt ""}} {
    set msg {[cgi_begin "Transaction Error"]
An error has occurred during the processing of your transaction.
It has been logged to our attention.<p>
$opt
[cgi_end]
}
    puts [subst $msg]
    syslog_log $log_list
    exit
}

# cgi_parse_querystring will typically be used to process something which
# has the form of a payment URL - there are signed name-value pairs
# in the query string.
#
# the query string is everything after the "?"
# the u_merchant argument is the name of an array to be filled in with
#   the principal record for the merchant who wrote the query string
# the u_secretkey argument is the name of an array to be filled in with
#   the right key for this merchant
# the u_fields argument is the name of an array to be filled in with
#   the name value pairs.
#
# in a case where there are multiple ?'s, cgi_parse_querystring throws
# away everything after the second ?
proc cgi_parse_querystring { query u_fields u_secretkey u_merchant} {
    upvar $u_secretkey secretkey
    upvar $u_merchant merchant
    upvar $u_fields fields
    global env
    # handle multiple ?'s, if any
    set first [split $query ?]
    if {[llength $first] > 1} {
        set query [lindex $first 0]
    }
    # parse the GET fields, if any
    if {[string length $query] > 0} {
        if {regexp {([^:]+):(.*)} $query dummy hash remainder} {
            cgi_parse $remainder fields
            if {!{pay_verify_signature $hash $remainder fields \
                    secretkey merchant]} {
                cgi_error cgi_parse_querystring "bad key: $query" \
                        "There was a problem with the URL"
            }
        } else {
            cgi_error cgi_parse_querystring "bad script arguments: $query" \
                    "There was a problem with the script arguments"
        }
    }
}

#
# cgi_fieldcheck takes an array and a list of required fieldnames
#   if any of the fieldnames are not defined as elements of the array
#   then cgi_fieldcheck returns 0
#
#   If all the fields exist, then each one is strimmed of whitespace
#   at the ends and any quotes are removed.
#   XXX Should expand to remove other bad stuff that might be in there
#
proc cgi_fieldcheck { a_fields fieldlist {missing xxx}} {
    upvar $a_fields a
    if {[string compare $missing xxx] != 0} {
        upvar $missing mlist
    }
    set mlist {}
    foreach field $fieldlist {
        if {![info exists a($field)]} {
            lappend mlist $field
            continue
        }
        set a($field) [string trim [string trim $a($field)] {\"}]
        if {[string length $a($field)] == 0} {
            lappend mlist $field
        }
    }
    if {[llength $mlist] > 0} { return 0 }
    return 1
}


#
# cgi_begin_nph
#
# deprecated
#

proc cgi_begin_nph { {title ""}} {
    return [cgi_begin $title]
}


proc cgi_title { t} {
    return "<html><head><title>$t</title></head>
            <body><h1>$t</h1>"
}

#
# cgi_begin
#
# transmits the stuff that you need at the beginning of a screen
# accepts an optional title
#
# text returned as a string, suitable for use in [subst] pages
#
```

```
proc cgi_begin { {title ""}} {
    global env
    if {[string first "nph-" [file tail $env(SCRIPT_NAME)]] == 0} {
        append res "HTTP/1.0 200 OK
Server: OMI/1.1
"
    }
    append res "Content-type: text/html

"
    append res [cgi_title $title]
    return $res
}

#
# cgi_end
# returns a signature line
#
proc cgi_end {} {
    global merchant_server_root merchant_demo_path
set msg {<p><hr><A HREF="$merchant_server_root/">
<IMG ALIGN="top" SRC="$merchant_server_root$merchant_demo_path/images/omicon32.g
<I>Copyright &#169; 1994 Open Market, Inc. All Rights Reserved.</I>
</body></html>
}
    return [subst $msg]}


proc cgi_return_error { {branch {}} { etext {}}} {
    set msg {[cgi_begin "Error"]
An error has occurred during the processing of your request.
It has been logged to our attention.<P>
[cgi_end]
}
    puts [subst $msg]
    syslog_log {list [list prog $branch] [list text $etext]}
    exit
}

#
# returns a really simple link
#
proc cgi_link { target text } {
  return "<a href=\"$target\">$text</a>"
}

proc cgi_form { target } {
  return "<form method=POST action=\"$target\">"
}

#
# cgi_self_link
#
# this is used to build links from a cgi script to itself which
# incorporate additional fields in a GET style
#
# cgi expects the name of "self" to be in env(SCRIPT_NAME)
# cgi expects "secretkey" to be an array with the appropriate key
#
#
proc cgi_self_link { {field_list {}} {a_fields xxx}} {
    global env secretkey
    upvar $a_fields fields
    # the following two statements create an empty array
    # in the case that match was not passed in
    set fields(xyzzy) plugh
    unset fields(xyzzy)
    foreach item [array names fields] {
        set nv($item) $fields($item)
    }
    foreach item $field_list {
        if {[llength $item] != 2} {
            error "Bad value passed to cgi_self_link $item"
        } else {
            set nv([lindex $item 0]) [lindex $item 1]
        }
    }
    set ticket [sec_create_ticket secretkey nv]
    return http://$env(SERVER_NAME)$env(SCRIPT_NAME)?$ticket
}

#
# stuff that was in http.tcl
# originally authored by A. Payne
#


# ------------------------------------------------------------------------
#
#  Return an "authorized required" reply to the client.  This will force
#  prompting for username/password.
#
proc cgi_return_auth_required {{realm "Open Market Account"}} {
set msg {HTTP/1.0 401 Unauthorised
Content-type: text/html
WWW-Authenticate: Basic realm="$realm"

[cgi_title "Authorization Required"]
Browser not authentication-capable or authentication failed.<p>
The username and password you entered were not
valid for access to the $realm.
[cgi_end]
}
  return [subst $msg]
}

# ------------------------------------------------------------------------
#
#  -- Returns an HTTP redirect to the client
#
#    Many clients have length restrictions on the redirect URL,
#    so we may need to return an intermediate page if the URL is
#    longer than what the client can handle.
#
proc cgi_return_redirect {url title desc linktext} {
    global env

    switch -glob -- $env(HTTP_USER_AGENT) {
        {NCSA Mosaic for the X Window System*}  {set maxlen 256}
        {Lynx*}                                 {set maxlen 1024}
        default                                 {set maxlen 128}
    }

    if {[string length $url] < $maxlen} {
        return [subst {HTTP/1.0 302 Found
Location: $url
Content-type: text/html
```

3 cgilib.tcl Mon Oct 10 17:33:44 1994

```
[cgi_title "Redirect"]
You appear to have a very old World Wide Web browser, that doesn't
support the redirect operation.  We strongly suggest upgrading to
the latest software.<p>
Here's the <a href=\"$url\">actual document</a>.
[cgi_end]]]}

    subst {HTTP/1.0 200 OK
Content-type: text/html

[cgi_title $title] $desc
<p><a href="$url">$linktext</a>
[cgi_end]}
}
```

5 cgilib.tcl Mon Oct 10 17:33:44 1994

```
#
#  Payment related library routines
#
#  L. Stewart
#  stewart@openmarket.com
#
# originally by payne

if [info exists env(SECRET_KEY)] {set secret_key $env(SECRET_KEY)}
if [info exists env(SECRET_KEY_ID)] {set secret_key_id  $env(SECRET_KEY_ID)}

# ---------------------------------------------------------------------
#
#  Payment stuff
#
#
#  Base link of all URLs referring to the payment system:
#
set paylinkbase "$payment_server_root/bin/nph-payment.cgi?"

proc paylink {args} {
    parseargs argvals {-text} $args
    set url [payurl $args]
    return "<a href=\"$url\">$argvals(-text)</a>"
}

# Create a payment URL.  This is used by paylink and others.

proc payurl {args} {
    global paylinkbase secret_key secret_key_id
    parseargs nv {-cost -url -domain -ttl -desc} [lindex $args 0]
    set fields(url) $nv(-url)
    set fields(amt) $nv(-cost)
    set fields(kid) $secret_key_id
    set fields(domain) $nv(-domain)
    set fields(expire) $nv(-ttl)
    set fields(desc) $nv(-desc)
    if {info exists nv(-id)} {set fields(id) $nv(-id)}
    set paylink [create-ticket $secret_key fields]
    return $paylinkbase$paylink
}

#
# Create an access URL
#

#
# expected fields: expire domain ip
proc pay_accessurl {url a_fields a_secretkey {fmt int}} {
    upvar $a_fields fields
    upvar $a_secretkey secretkey

    set ticket [sec_create_ticket secretkey fields]

    switch -- $fmt {
        int {
            regexp {([^\/]+\/\/[^\/]+)\/(.*)} $url dummy front end
            set result $front/0$ticket/$end
        }
        get {
            set result $url?$ticket
        }
        default {
            cgi_error "pay_accessurl unknown format $fmt"
        }
    }
    return $result
}


# ---------------------------------------------------------------------
#
# construct a smart link to the thing purchased, for purposes
# of putting into a smart statement
#
proc pay_build_smartlink {domain url expiration tid a_secretkey fmt} {
  global env
  upvar $a_secretkey secretkey

  if {$domain != ""} then {
      set nv(domain) $domain
      # origin=ss means this URL came from the smart statement
      set nv(origin) ss
      set nv(expire) $expiration
      set nv(ip) $env(REMOTE_ADDR)
      set nv(tid) $tid
      return [pay_accessurl $url nv secretkey $fmt]
  } else {
      return $url
  }
}

#
# build a keyid suitable for transmission
#
proc pay_makekeyidpair { mid kid } {
    regsub {[.]0$} $mid "" mid
    regsub {[.]0$} $kid "" kid
  set result $mid.$kid
}


# Parse an argument vector.
# Usage: parseargs array-name list-of-options arg-vector
#   array-name is the name of an array that will get the options
#   list-of-options is a list of option names
#   arg-vector is the argument list to be processed.

proc parseargs {name argnames arglist} {
        upvar $name a
        foreach i $argnames {
                set a($i) ""
        }
        for {set i 0} {$i < [llength $arglist]} {incr i} {
                set opt [lindex $arglist $i]
                incr i
                set val [lindex $arglist $i]
                set a($opt) $val
        }
}


# ---------------------------------------------------------------------
#
#  Verify merchant stuff:
#       - valid merchant ID
#       - valid merchant signature on payment URL
#
#  Any errors here are fatal (i.e. someone's been mucking with the URL)
#
```

1 payment.tcl Fri Sep 16 09:23:17 1994

```
proc pay_verify_signature {hash remainder a_fields a_secretkey a_merchant} {
    upvar $a_fields fields
    upvar $a_secretkey secretkey
    upvar $a_merchant merchant
    global env
    if {![info exists fields(kid)]} {
        cgi_error verify_signature "no $kid: $env(QUERY_STRING)"
    }
    set kid_split [split $fields(kid) .]
    if {[llength $kid_split] != 2} {
        cgi_error verify_signature "kid_split bad format: $env(QUERY_STRING)"
    }

    set mid [lindex $kid_split 0]
    set kid [lindex $kid_split 1]
    if {[db_read_record secretkey secretkey_id $kid secretkey] != 1} {
        cgi_error verify_signature "bad key $kid: $env(QUERY_STRING)"
    }

    if {$mid != $secretkey(principal_id) } {
        cgi_error verify_signature \
                "merchant keyid mismatch: $env(QUERY_STRING)"
    }

    if {[db_read_record principal principal_id $mid merchant] != 1} {
        cgi_error verify_signature "bad merchant $mid: $env(QUERY_STRING)"
    }
    set signature [md5 "$secretkey(secret_key) $remainder"]
    if {[string compare $hash $signature] != 0} {
        cgi_error verify_signature "bad signature: $env(QUERY_STRING)"
    }
    return 1
}

#
# get user name and password
#

proc pay_getuser {a_user {realm "Open Market Account"} } {
    global env
    upvar $a_user user
    if {![info exists env(REMOTE_USER)]} {
        puts [cgi_return_auth_required $realm]
        exit
    }

    if {![get_principal $env(REMOTE_USER) user]} {
        syslog_log [list pay_getuser \
                [list "Invalid user id" $env(REMOTE_USER)]]
        puts [cgi_return_auth_required $realm]
        exit
    }
    if {![info exists env(REMOTE_PASSWD)]} {
        syslog_log [list pay_getuser \
                [list "no REMOTE_PASSWD"] \
                [array_to_list env]]
        puts [cgi_return_auth_required $realm]
        exit
    }
    if {$env(REMOTE_PASSWD) != $user(access_password)} {
        syslog_log [list pay_getuser \
                [list "Invalid password" \
                " for $env(REMOTE_USER): expected '$user(access_password)', \
                got '$env(REMOTE_PASSWD)'" ]]
        puts [cgi_return_auth_required $realm]
        exit
    }
}


#
# copy key fields from principal record to secretkey record
#
proc pay_prin_to_key {a_prin a_key} {
  upvar $a_prin prin
  upvar $a_key key
  set key(principal_id) $prin(principal_id)
  set key(secretkey_id) $prin(secretkey_id)
  set key(secret_key) $prin(secret_key)
}



# duplicate_check scans the duplicate table for any
# record where the initiator matches cid and the
# benificiary matches mid and the domain matches domain
# and the access has not yet expired.
proc duplicate_check { cid mid domain } {
  global syb
  set now [currenttime]
  case [execsql "select transaction_log_id from duplicate
    where (initiator = $cid
    AND benificiary = $mid AND domain = \"$domain\" AND
    expiration > $now)"] {
        {NO_MORE_ROWS} {
            return 0
        }
        {REG_ROW} {
            return [sybnext $syb]
        }
    }
}


#####
#
#####

# Enter a transaction in the database
#
# cid: numeric customer id
# mid: numeric merchant id
# amount: money
# ipaddr: format xx.xx.xx.xx as a string
# domain: varchar(40)
# expiration: a delta interval in seconds
# url: varchar(255)
# description varchar(40)
#
# at the moment, the only transaction code is "   p" for payment
# this will be extended for returns, disputes, preauth, credit, etc.

proc enter_transaction {initiator benificiary from_account to_account
  amount currency ip_address domain expiration url description date fmt} {
  #
  # create record for transaction_log
  #
  set t(amount)              $amount
  set t(currency)            $currency
```

```
set t(transaction_date)   $date
set t(initiator)          $initiator
set t(beneficiary)        $beneficiary
set t(from_account)       $from_account
set t(to_account)         $to_account
    if {[string compare $fmt "get"] == 0 } {
        set t(transaction_type)  "g  p"
    } else {
        set t(transaction_type)  "   p"
    }
set t(ip_address)         $ip_address
set t(domain)             $domain
set t(expiration)         $expiration
set t(url)                $url
set t(description)        $description
#
# insert record into transaction_log
#
set tid [insert_row transaction_log t]
#
# now put the requisite information into the duplicate table
#
set d(transaction_log_id) $tid
set d(initiator)  $t(initiator)
set d(beneficiary)  $t(beneficiary)
set d(domain) $t(domain)
set d(expiration) [expr $t(expiration) + $date]
insert_row duplicate d
regsub "\.0$" $tid "" tid
puts $tid
```

5 payment.tcl Fri Sep 16 09:23:17 1994