A Multiprocessing Computer System Composed of Loosely-coupled FORTH Micro-chip Modules

Gabriel Castelino and Richard Haskell
School of Engineering & Computer Science
Oakland University, Rochester, Michigan 48063

INTRODUCTION

A Multiprocessing Computer System consisting of loosely-coupled FORTH micro-chip modules is currently being developed at the School of Engineering and Computer Science at Oakland University. A network of 16 modules has been designed. The modules do not share any common memory and communicate with each other over a global bus -- an 8-bit parallel bus with 7 control lines. The system uses a layered protocol structured on the OSI Reference Model [1]. The network is designed to support different types of micro-chips operating at different speeds. The software support at the physical and data-link layer will have to be modified for the different types of micro-chip modules.

In such a system, it would be desirable for one module to execute a word that has been defined in another module in the system. This is referred to as a Remote Procedure Call (RPC). In the system being developed calls of this type are handled by the RPC layer which sits above the Data-link layer of the OSI model. The FORTH words defined at this level are independent of the type of micro-chip being used.

Example of a Remote Procedure Call

A brief example will set the stage for the discussion that follows. Assume two modules with identification numbers 6 and 8 connected on the network. The user on module 6 would like to execute a word EXAMPLE in module 8. The user declares EXAMPLE to be a remote word (word defined in some other module) using the word REMOTE as follows:

```
8 REMOTE EXAMPLE.
```

After the above declaration, EXAMPLE can be used in the normal FORTH manner and the fact that it is a remote word is now transparent to the user. EXAMPLE is now part of the vocabulary in module 6 with a link to its counterpart in module 8. This link is established by the RPC layer as will be discussed in the sections that follow.

IMPLEMENTATION

The following sections will discuss the implementation in FORTH of the RPC layer. The source listings of some of the words appear in the Appendix.

Definitions

Caller - The module initiating the remote call.
Callee - The module being called.

The protocol at the RPC layer uses the following fields

<callee id> <caller id> <entry#> <#bytes> <parameters>

where,

<callee id> is stored in the variable RECID, **<caller id>** is the constant MYID, **<#bytes>** is stored in the variable #BYTES and tells the callee how many bytes follow, **<entry#>** is the entry

number of the word being referenced and is stored in ENTRY.

Each of the above fields is 1-byte long. This implies that the number of parameters is limited to 256 bytes or 128 words. The identification (id) numbers of the modules are assigned arbitrarily. The determination of <entry#> corresponding to a particular word will be discussed in the following sections.

Primitive Words

The following words are defined at the Physical and Data-link layers and are dependent on the type of micro-chip used.

SEND - send one byte of data to callee.

RECEIVE - receive one byte from caller.

CALL - Establish link between caller & callee. Uses <callee id> from RECID. Returns 0 if successful. If an error occurs the error number is stored in the variable ERRNO and a non-zero value returned.

.ERROR - Prints error message corresponding to error stored in ERRNO and aborts.

ENDCOM - Terminate link.

Remote Words and Entry Numbers

All words in a module that may be accessed by other modules are stored in a table called ENTRIES as follows:

```
[WORD0] #in0 #out0
[WORD1] #in1 #out1
[WORD2] #in2 #out2  ...
```

where [WORD1] is the cfa of WORD1; #in1 and #out1 refer to the number (in bytes) of input and output stack items respectively. The <entry#> referred to earlier is the index of the word being referenced in the table ENTRIES.

WORD0 is a special word called RET.ENTRY that accepts an input word and returns the entry# (index) and #in corresponding to the word (returns a 0 if word is not in the table).

The REMOTE Declaration

The caller module identifies a word as being remote by using the word REMOTE as follows:

```
<callee id> REMOTE <word>
```

REMOTE executes GET.ENTRY which makes a remote call to the module identified by <callee id>. The <entry#> is 0 which causes RET.ENTRY to be invoked in the callee, thus returning the entry# and #in of <word> as results of the call. After REMOTE executes (compile-time) it forms a dictionary header for <word> with its parameter field containing <callee id> <entry#> <#in>.

To execute the remote word the user executes <word> in the normal FORTH manner. This causes a remote call via RPC.SEND. It is important to note that the determination of <entry#> is done during compile-time. At run-time <entry#> is available in the parameter field of the corresponding word. <#in> tells RPC.SEND how many parameters to read off the stack. It may also be used to check for sufficient number of stack items before initiating the call.

More about ENTRIES

The table ENTRIES is created by the defining word RPC.TABLE as follows:

```
<#entries> RPC.TABLE ENTRIES <word1> <#in1> <#out1>
                             <word2> <#in2> <#out2> ...
```

Any word in the table may be executed by referring to its index (<entry#>) in the table. Executing ENTRIES with <entry#> on the stack will execute the corresponding word. This is how RPC.RECV executes a remote call in the callee module. All the words in the table may also be executed locally in the normal FORTH manner. Thus the remote user and the local user both execute the word in the same manner.

Replying to a Remote Call

After a remote call has been received, the word corresponding to the <entry#> is executed. The results are returned to the original caller (who now becomes the callee) using the same protocol. The received <entry#> is used with bit 7 set to 1 to indicate that it is a reply to the remote call corresponding to <entry#>. This limits the number of words that can be referenced remotely to 127.

CONCLUSION

The RPC layer implementation described in this paper provides a means by which a remote procedure call is transparent to the user. The higher layers of the system can use the RPC layer to develop a very user-friendly multiprocessing environment. The scheme discussed here may be used with other networks provided the necessary changes are made to the primitive words at the physical and data-link layers.

REFERENCES

[1] Zimmerman, H. "OSI Reference Model - The ISO Model Architecture for Open Systems Interconnection," IEEE Transactions on Communications (COM-28)4, Apr.1980, 425-432

Appendix

The following is a source listing of some of the words in the implementation of the RPC layer. The words **@RECID, @ENTRY, @#BYTES** and **!RECID, !ENTRY, !#BYTES** fetch and store the contents of the variables **RECID, ENTRY and #BYTES.** The word **WAIT.REPLY** waits for a reply from the called module. The reply is handled by RPC.RECV.

```
( Scr# 23                rpc3.2                              gsc)
( n items -- | send n bytes after establishing link)
( the value n is stored in variable #BYTES)
: SEND.MESS
   CALL IF .ERROR
        ELSE MYID ( caller id) SEND @ENTRY ( entry#) SEND
             @#BYTES ( #bytes) DUP SEND
             0 DO SEND LOOP ENDCOM ( terminate link) THEN ;
( Receive & execute a remote procedure call)
( Or receive reply - entry# > 128)
: RPC.RECV
   REC.MESS ( receive parameters from caller) @ENTRY
   DUP 128 < ( reply to my call ?)
   IF ( no) DUP 128 + !ENTRY ( set bit7 of entry# for reply)
      ENTRIES ( execute word) SEND.MESS ( send reply)
   ELSE DROP THEN ;
```

```
( Scr# 24                rpc3.2                              gsc)
0 VARIABLE RPC.BASE  ( base address for table)
: ERR  0 !#BYTES ;
( -- cfa | returns the cfa of next word in input stream)
: FIND  [COMPILE] ' CFA ;

( This is entry 0 in table ENTRIES)
( -- #in\entry# | return entry# of word in TIB)
: RET.ENTRY
   0 IN ! FIND ( cfa of word) RPC.BASE @ ( base addr. of table)
   DUP 2+ SWAP @ ( #entries) 1+
   1 DO 4 + 2DUP @ - 0= ( cfa's equal ?)
        IF DUP 2+ C@ ( #in) I ( entry#)
        ROT >R ROT R> ( 2swap) LEAVE THEN
     LOOP @ - ( not found ?) IF ERR THEN ;

( Scr# 25                rpc3.2                              gsc)
( store WORD0 in table )
: WORD0  ' RET.ENTRY CFA , -1 C, 2 C, ;
( -- n | get number from input stream)
: GET.#  32 WORD HERE NUMBER ( double precision) DROP ;

( entry# -- )
( <#entries> RPC.TABLE <name> <word1> <#in> <#out> ...)
: RPC.TABLE
   <BUILDS DUP , WORD0 0 DO FIND , GET.# C, GET.# C, LOOP
   DOES> DUP RPC.BASE ! SWAP 2DUP SWAP
         @ > OVER 0< OR   ( check range)
         IF 2DROP ERR
         ELSE 4 * + 2+ DUP 3 + C@ ( get #out) !#BYTES
              @ ( get cfa) EXECUTE ( execute word) THEN ;

( Scr# 26                rpc3.2                              gsc)
( get next word in input stream to HERE without moving pointer)
: GET.WORD  IN @ 32 WORD IN ! ;

( -- #in\entry# | get entry# for word at HERE from remote mod.)
: GET.ENTRY
   CALL IF .ERROR
        ELSE MYID ( caller id) SEND 0 ( entry#) SEND
             HERE COUNT ( #bytes) DUP SEND
             0 DO DUP C@ SEND 1+ LOOP
             DROP ENDCOM WAIT.REPLY THEN ;

( Scr# 27                rpc3.2                              gsc)
( addr -- |send remote call/callee id, entry# & #bytes at addr)
: RPC.SEND
   DUP C@ !RECID 1+ DUP C@ !ENTRY 1+ C@ !#BYTES
   SEND.MESS WAIT.REPLY ;

( <recid> REMOTE <word> |identify a word as a remote procedure)
: REMOTE
   DUP !RECID GET.WORD GET.ENTRY
   <BUILDS ROT C, C, C,
   DOES> RPC.SEND ;
```