Final Project Report

for

Advanced Microprocessor Design
CS 6825

Due 23, 1996

Finished By:
Zhao chengyan 255459

Finished By:
Wang yi 252370

We hereby claim that this project is finished by ourselves only and not any portion of which is copied from any other people.

Index

Auxiliary Processor - AP

With

Parallel Processing in two 68000 ECB boards

CS6825 Advanced Microprocessor Systems

Fall 1996

TERM PROJECT:

Prepared for Dr. Pochec

By : Zhao Chengyan & Wang Yi
(Graduates of CS6825)

The University of New Brunswick
Faculty of Computer Science
Fredericton, NB
Dec 23, 1996

2. Project Requirements & Objectives

1>. Design requirements

Design and implement a real time programming kernel for emulating auxiliary processor (coprocessor) functions on a second MC68000 ECB, called here AP, that will allow to execute operations on a remote computer.

The top view of the whole system

1). Master & Slave connected in port 2

2). Both program & control code would transfer through serial port 2

Description:

1>. The objective of this project is to allow to run programs on two computers simultaneously. Control over the programs is to be provided through two mechanisms:

2>.Calls to auxiliary processor initiated by unimplemented instruction exceptions: For example, use a line A exception to implement the following operations:

1)>. OPERATION1: submit a task for execution on auxiliary processor; specifies block in memory where the relocatable code and data reside and transfers the code to the auxiliary processor.

2)>. OPERATION2: check the status of the job on the auxiliary processor.

3)>. OPERATION3: get results back form the auxiliary processor (you may transfer the entire block back including the code and assume that the data area contains the results); specifies where to in the host memory the results are to be transferred .

3>. Real time monitor commands, for example:

For the host: same as operations 1, 2, 3 above.

For the auxiliary: same as operation 2 above.

The Auxiliary Processor programming kernel is to be implemented as two parts: (I) kernel for the host processor, and (ii) kernel for the auxiliary processor.

Procedure:

Define the requirements for the auxiliary processor calls. State the function supported by each call, parameters needed etc. Prepare detailed high level specifications for the communication protocol between the host and the auxiliary processors. Select a subset of real time monitor commands to be implemented on the host and on the auxiliary processor.

Design and implement a real time kernel for each of the processors. Use the monitor commands on the host to run a sample program on the auxiliary processor. Program a sample MC68000 applications that includes calls to the functions to be executed on the auxiliary processor. (In addition to MC68000 application you may try an IBM PC application (use programming language of your choice) that talks to the auxiliary processor through a serial port.

2). Requirements:

Prepare final report and a disk with the programs developed for this project. Final report should include: description of the problem, high level specification, motivation and description of the real time monitor functions and the functions accessible through the unimplemented instruction exceptions, detailed description of the design including the source code, description of the testing procedure, and finally a short instruction manual that would allow a CS4825 student to run a sample application.

Also include an estimate on time (hours) it took to complete this project. If you decided to work on different modules individually (e.g. one student develops kernel for the host the other for auxiliary processor) please specify your contributions.

Due date:

December 23, 1996

3. Top Design:

1). Requirement for top view & objective:

The objective of the project is to show a task initiated from one node can be carried out on another node. The status of the execution is monitored, and the results expected can be achieved promptly.

1>. Submit an application in an un-implemented exception from the host(in our case, the exception would be a Line A exception, with the parameters transferred in the D0 register).

2>. Slave will receive the application from the host and save it in a certain place in its private memory(relocatable).

3>. Both master and slave will response for the user (PC keyboard): it must have a KBD response module.

4>. After the computation (execution of the submitted task) finished, the slave will return the results to master.

5>. Also, the master will receive the results coming back from slave and save the results into its private memory.

2). Kernel in the Master ECB:

1>. Send the task to the slave:

Send a task execution code (bin code) ready in the memory to the slave ECB board.

Strictly speaking, this modular is not under the control of timer dispatcher. Whenever it begins to execute, it would disable the time and after it successfully finishes, it would enable timer again.

2>. Inquiry Keyboard & response:

The user in the master ECB control computer has the ability to inquire the application status. This is supported from a KBD response modular in the master ECB, it is specially designed to response the master user inquiry.

3>. Get results back:

This modular is used to get back the results of the submitted application to the remote ECB(auxiliary computer). This is required because the user (in master ECB board) want to know the result of the remote executed application.

There would be other user applications in the master ECB computer, since the remote execution could happen simultaneously with the current ECB computation.

3). Kernel on the slave ECB:

1>. Receive App:

The receive Application modular will receive the application submitted from the master ECB.

Whenever the modular is active and established connection, it would disable the timer, receive the whole message block, append the task termination programs in the end of the received application, change the status of its own into #BLOCKED, change the status of the executable application (to be #READY), reenable the timer and go into an infinite loop, waiting for the timer dispatching

2>. Execution application modular:

This modular is used to remotely execute the application received from the master ECB. Whenever the timer dispatches it, it would have ability to execute. When it finishes running, the program control will go to the self-appended part, which will change the status and block itself.

3>. Inquiry and KBD response modular:

This modular is quite similar as the one in master ECB, with the following difference:

1)>. It would response the inquiry from the master ECB (Reply BUSY).

2)>. It would response the keyboard inquiry from the slave PC keyboard and display the current status of the task.(BUSY or IDEAL).

4>. Result back:

This modular is used to send the results back to the master.

The whole required results are the D0..D7 eight data registers, which contain the results as the master requires.

4). Design of communication & control protocols

1>. Source program & data submitted from master:

The program & data submitted from master is composed of :

1)>. Length of program.

2)>. Program execution code(bin code).

/* for a more detailed description, please see picture 6).

An improvement from the presenetation:

We noticed that the program would init the data register it would use, so there would be no reuqirement for the program to submit data. So, we delete the data part in the submitted program.

2>. Return results:

The returned program results are the contents of the eight data registers(D0..D7): Picture 6

1)>. Because the length of the result is certain(32 bytes, 8 long words), we define the result length previously and there would be no requirement to submit the length.

2)>. The results generated from slave program are stored in memory and will be transfered back to master from the serial port(ACIA2) one BYTE a step.

3>. Protocol control: handshaking signals

It is required to do the hand shaking processing before the communication. A special kind of handshaking signals are used to do this job (before the beginning of any type of communication).

4. Implementation

1). Implementation on the Master ECB board

1)>. Objective in master ECB

All requirements on master ECB could have three basic functions The task transfer from the host to the slave, the task execution status monitor, and the task result collection.

The task switching is done through the timer interrupt and will be the same as that for the AP. It will not be elaborated here but in the AP implementation section.

2)>. Sending the program to slave: Implemented in Line A exception

Line A exception is used to implement the program transfer to the AP. Like any other exception, the exception service routine address should be specified at the address corresponding to the Line A exception vector ($28). As a software initiated exception, an illegal instruction with the first four bit as "1010" is used to trigger it. In the implementation, a jump (JMP) is made to the illegal instruction in the memory. Because the illegal instruction is never executed actually, the system stack will contain the PC of the JMP. RTS will restore the JMP instruction, then another line A exception will appear. To prevent this, the stack is reloaded with the PC where the termination of the sending task is stored.

3)>. Receiving results from slave:

Result receiving and execution monitoring is done by using the ACIA2 to receive and send the required signal. Nothing special.

4)>. Task switching mechanism:

Since the three operations are mutually exclusive. At one period, only one task is activated. The deactivation and activation are done by changing the status of the task at the end of each operation. The selection of task is easy because only one task will be executable at one time. By comparing the status word, the active task will be executed. The task switching for the host is to change the operation mode rather than to make concurrent running of tasks. The basic operation can be expanded so that each operation can be executed concurrently with other tasks. The more advanced system can be easily implemented by adding task switching functionality for concurrent running.

5>. Program flow:

The communication protocol defines the behaviour at the protocol level for both the message exchange between host processor and the AP, the program flow inside each processor is:

A. Composing the application program

B. Enable Real Timer Task Switching

The only active task at the initial time is the sending of the application program to the AP. The sending application program task will be blocked after mission completed. The keyboard response will be active upon the completion of the sending application program task.

C. When the communication request message (result available) is received from the AP, the keyboard response is blocked, and the receiving results task is unblocked, being the only active task at that moment.

6>. Communication flow control:

The communication mechanism plays one of the key roles in the project. The task transfer, monitor and results collection are done through the serial communication port. For this project, two aspects are to be considered, the lower level is the communication link setup (handshaking), the high level is the protocol above the data link layer. The handshaking procedure can assume the useful data to be transferred at the right time, thus prevent useful data from missing at the receiver and prevent the useless data to be collected.

The high level protocol guides the whole procedure to be operated smoothly and prevents deadlocks that usually occur in the communication systems.

1>. Handshaking:

Before the useful data can be transmitted, a procedure called handshaking is needed. Because the ACIA2 does not provide the necessary flow control functions and enough buffer size at the data link level, some of the data transmitted when the receiver is not ready will be missed. On the other side, the buffer makes it possible useless data may be collected by receiving party. The only way to guarantee the success of data transmission is to make both parties ready at the same time and clear the useless data in the buffer before the useful data transmission. So each byte transmitted will be collected at the other side immediately and correctly. A handshaking started at the sending of communication inquiry signal(the "I") from transmit party and ended with the acknowledgment signal (the "o") from the receiving party. The useful data will be transmitted provided that the receiving of "o" at the transmit party.

After handshaking, the transmit party is ready to send data and the receiver party is ready to receive data while the buffer is cleared.

2>. Communication protocols:

The program flow in both the host and the AP determine the requirements for the protocol. The protocol should guide the program flows smoothly and prevents errors. When the expected data can not be received, a deadlock occurs. So the design of the program control of both the host and AP should estimate all the possible signals to be received at any state. For example, usually a handshaking signal from one side "I" (initiate) should expect a "o"(ok) from the other side, however, this is not true always. A "I" transferred from the host will not be received or should be accepted and processed at the AP when AP has finished the calculation and tries to initiate the result transfer back signal. Rather than "o", a signal called "I" is received at the port. If host does not consider this case, AP will ignore the "I" and wait for the never coming "o" for ever. Fortunately, the system does not require many complicated signal transmission and the system does not have many states. So it is easy to identify all the possible signals at all the stages. Following is all the communication occurs during the operation.

```
         i
MP ---------------------------> AP    /* this is the handshaking part of the message transferring */
         o
   <---------------------------
         program
   --------------------------->          /* this would be the source program transferring to AP */


           I
MP -------------------------> AP    /* this is the handshaking part of MP inquiry */
           o
    <-------------------------
           q
    ------------------------->          /* this is the request information and following is the request result */
           n
   <--------------------------


             I
MP <------------------------- AP           /* handshaking signals to get results back */
             o
   -------------------------->
           result
  <--------------------------

              I
```

```
host <------------------------AP          /* after the handshaking , the results would transfer back */
            o
   ------------------------->             /* to the master */
           result
   <-------------------------
```

Because the control states of each processor are explicated defined, there will be no requirement to add an identification and addressing tag to each message body. Most of the messages are mainly composed of a byte, except the message containing the application program in which the length of the application program is to be specified. For complex system composing of more than one AP or requiring more than one tasks to be processed in one AP, the message encoding should be carefully defined and the protocol control part will be a little bit more complex.

2). Implementation on Slave ECB :

1>. Main frame:

Both the program developed in Master & slave ECB are in the control of the main frame. That is: the multi-tasking control main frame.

The main frame does everything that a multi-tasking system requires, as:

1)>. TCB control:

Including: TCB initialization, TCB save (save the current CPU working environment into TCB correspond), TCB restore (restore the selected TCB contents into CPU registers).

2)>.Multi-tasking dispatching:

The multi-tasking dispatching algorithm is a little bit complex, it would consider:

A. The time slice left of the current TCB

B. The current TCB task status (READY, RUNNING, BLOCKED)

C. Dispatching selection algorithm:

If a task has to be dispatched, the selection algorithm would try to find the next available TCB (status is READY, have execution time, etc). And try to restore the selected task into CPU.

/* For a more detailed description and source code reference, please view the source code provided in timer.x68, starting address is : $4600 */

2). Application receive modular: it starts at physical address $2500

1>. Try to see if there is the connection request signal.

2>. If there is such signal in the serial port 2. Reply to the request a ‘o’ Signal to establish the connection, and disable the timer.

3>. The first byte is the submitting program size. Due to this design, the size of application submitted would be limited to 255 bytes in length.

4>. Try to receive the whole message block, with the length identification at the very beginning of the message.

5>. Do self-programming at the end of the receive application:

1)>. Change the status of this modular in the TCB status field.

2)>. Active the “execution application” modular: TCB status field to : READY.

3)>. Reenable timer;

4)>. Go into an infinite loop to wait for the timer dispatching.

3). Slave KBD response & Inquiry response modular: is starts at address $5A00

1>. If there is inquiry message coming from the master ECB ready in the serial port 2, this modular has the responsibility to reply the status of the submitted task. (Since it is still executing, the reply message would be “n” for ever)

2>. If there is inquiry from the slave ECB connection computer keyboard, it also has the responsibility to reply to this user. (Response the “Q” as “BUSY”)

3>. The status of the task is saved in a status word buffer, which is hold by the slave ECB only.
4>. The status word indicates the status of the current executing task and only could be changed if the task is finished executing.

4). App execution modular: it starts at physical address $2900

1>. When timer dispatch comes and the submitted App is ready, it would gain the control and begin to execute.

Please note: in order to have a better execution performance, we have the following static time slice allocation:

Receiver : 1 time slice;
Sender : 1 time slice;
KBD response: 1 time slice;
Execution: 5 time slice;

2>. Before it stops, there would a long jump in the source bin code, jump to the "**end task**" block to finish this execution

3>. In the **"end task**" block, it would do several things:

1)>. Save the current working Data registers into memory as results;
2)>. Block the execution task and set the sender task to be ready;
3)>. Reenable timer.
4)>. Go into an infinite loop, waiting for the timer to dispatch;

5). Result send modular: it starts at physical address $2700

1>. Try to get connection with MP by sending the "I" signal to port 2

2>. If a reply to of 'o' comes back, the connection is established in both. Disable the timer to prevent additional, unexpected dispatching.

3>. Submit the 32 bytes of results (saved into consecutive memory location) to the master ECB consecutively. Because the length of results is known before transferring , there would be no additional identifer to indicate the length of results..

4>. Change the task status:

1)>. Set the status of the "sender" task in AP to be "BLOCKED";
2)>. Also block the KBD response task.
3)>. Call the 228, Trap #14 to stop

6). Message transferring & flow control:

The same as those of the MP's.

For a more detailed explanation & discussion, please view **the program flow control** in Master ECB implementation part:

5. Possible problems, programming & debugging experience:

1). Multi-tasking & non-multitasking:

The multi-tasking is available during the submitted program executing time.

In the time that the receiver begins to receive and the sender begins to send and the time to response the KBD inquiry(no matter where it is from), the timer would be disabled, in order to avoid the un-expected job dispatching and loss of data in serial port.

2). Long jump in the submitted App:

There is an absolute address long jump (JMP $2600) in the end of the submitted App. Following is the explanation:

1>. The absolute address contents are reserved to do the "end_of_execution" work for the submitted

App. I have tried several times to append the same kind of code to the end of the submitted program when finish receiving, but it failed. Moreover, we found the limitation:

1)>. The appending address must be an even address;

2)>. The appending address must be an 4's time address;

If any of the two factors is not true, the appended program would be corrupted in the very beginning. This is a limitation of the 68000 instruction set which requires to start at the even address.

2>. Solutions:

1)>. The sender of MP must be sure that the program sent is satisfied by the regulation above. If not, one or more (up to three) NOP instruction(s) would be inserted into the end of the program.

2)>. The submitted App would finish with a long JMP instruction, jumping to the code to deal with the program appending.

It is clear that the second solution is easier and simpler. We choose the second solution in our project, although with some kind of sacrifice of the program relocation.

3). Possible screen display block:

There is an display bug in the slave program. When the user in Master try to make a lot of inquiry in a very short time, the master would try to send a lot of “I”s consecutively. If the speed of sending inquiry exceeds the speed of the serial port, something unknown would happen. The slave display task would not be able to display the current execution task properly, but the execution will have no affection. The program could finish execution and send results back correctly and stop smoothly.

A suggestion: please do not press the inquiry button consecutively without releasing the key in the master keyboard.

6. Dedication of different people

1). Wang’s dedication:

1>. Master ECB sender modular
2>. Master ECB receiver modular
3>. Master ECB KBD inquiry & response modular
4>. Simple program designed to remote execute
5>. Simple multi-tasking algorithm & control;

2). Zhao’s dedication:

1>. Multi-tasking main frame, with the complex task dispatching algorithm’s implementation
2>. Slave ECB receiver modular
3>. Slave ECB execution modular
4>. Slave ECB sender modular
5>. Slave ECB KBD response & remote inquiry response
6>. Final project report

3). Public dedication:

1>. The message buffer format
2>. The format of submitted app’s extension
3>. Handshaking signals format & flow control

7. Project time

1). Top design time: 10 hours
2). Programming & debugging time in separate board: 10 hours
3). Handshaking & communication in both board: 15 hours
4). Prepare code & final report: 8 hours
5). Others: 5 ~ 10 hours

Total time consumed : 50 hours

8. Testing & Demo show:

1). Testing in Master ECB:

1>. Trace test on sender modular: to be sure the handshaking and app bin code sending could finish smoothly.

2>. Trace test on receiver modular: to be sure the handshaking and app results could come back correctly.

3>. KBD modular test: to be sure the inquiry message could transfer back and forth smoothly and the replied message results could display onto screen correctly.

2). Testing in Slave ECB:

1>. Trace test on the receiver modular: to be sure that the handshaking and app bin code receving could finish correctly.

2>. Trace test on the sender modular: to be sure the successfully established connection and smoothly sent application execution results;

3>. Execution test: the simple execution test to see that the submitted task would have the same results as it works in the master ECB.

4>. KBD response test:

1)>. Test the modular could response the inquiry from slave Keyboard correctly.

2)>. Test the modular could response the inquiry from remote ECB (Master) correctly.

3). Demo Show:

1>. Make hardware connection:

Please make the hardware connection as described in the Objective part of this report.

Please note: the connection between the two ECBs must be made on port 2(the port on the right) and a null modem is required.

After the hardware connection, please use the TM(Transparent) command in pcplus to test if the hardware connection is established.

1>. Load program:

The program executes in Master ECB is : host1.rec, while the program executes in Slave ECB is called: timer.rec

Please load the two programs into different ECBs separately. There is no limitation on which ECB to be the Master or Slave. It depends on what program loaded.

2>. Set Program Counter:

Because the very beginning part of the loaded program is the data section, the PC must be reset to be able to execute.

In both ECBs, set the PC to 2000 physical address.

e.g. : **.PC 2000 <return>**

3>. Run program:

1)>. In Master ECB, press: **G <enter>**

2)>. In Slave ECB , press: **G <enter>**

Note: there is no "begin execution" sequence requirement. Any board could start at any time.

4>. Monitor execution:

1)>. In Master ECB:

While the program is submitted and executing remotely, there would be no echo on Master ECB. To inquiry, press 'q', the reply would be : "calculating"

2)>. Whenever the program is received and app execution begin, there would be a hyphen ("-") to indicate the app is executing;

Local inquiry: press "Q", a reply "BUSY" would show to indicate the current executing task status;

Remote inquiry: whenever there is a remote inquiry, a "INQUIRY" would display onto the Slave ECB's screen.

5>. Read results:

When the execution & results transfering finished, both the program would stop smoothly by calling the Trap #14 function 228.

The result in Master would be saved into : $6500, 20 byte in length

The result in Slave would be saved into : $1806, 20 byte in length;

to view the results , **MD 1806 20 <return>** (In Slave) and **MD 6500 20 <return> in Master**

9. Appendix

1). Master Program: host1.x68 with comments on file (printed separately)

2). Slave Program: timer.x68 with comments on file (printed separately)

3). Disk with developed programs:

The program with the name above are all saved in the root directory.

Together with the MC68000 execution file, with the extension of .REC