IOP Manager ERS

G D

Rev. 1.0 A1 1/18/89

Introduction

The Modern Victorian architecture, and Four Square and F19 implementations, contains two Input Output Processors (IOPs), formerly called Peripheral Interface Controllers (PICs) which are programmable input / output processors that have a shared memory interface with the main CPU (68030). By off loading some of the input / output tasks to the IOPs, the main CPU will have more free cycles and better performance in a multitasking environment. This document will describe the method of passing messages between the main CPU and the IOPs, and other IOP related functions. Specific information about how certain drivers interpret the contents of these messages are covered in separate documents. Much of the information in this document is very technical and detailed, and is provided for mainly for those people who will be writing and debugging drivers on the IOPs. This document will also serve as the design specification for the IOP Manager.

Hardware Overview

An IOP is a single chip computer consisting mainly of a 65CX02 processor running at a clock rate of 1.9584 MHZ (≈ 510 ns.). one 16 bit timer/counter, two DMA channels. and 32K bytes of external shared RAM. The 680XX CPU can read or write to any byte in the IOP RAM address space. The IOP cannot directly access any of the 680XX memory address space. There are cross interrupt lines that allow the 680XX to generate a single interrupt on each IOP, and each IOP can generate two different interrupts on the 680XX. On the current IOP based CPU projects, one IOP will be connected to a SWIM disk interface chip. This IOP will contain the code to support the SONY Driver, and the Apple Desktop Bus (ADB) Driver. The other IOP will be connected to the SCC chip, and will contain code to support the Serial Driver, and the Apple Talk Drivers. The existing 680XX SONY, Serial, and Apple Talk Drivers will be modified to communicate with the driver code on the IOPs, by using calls to the IOP Manager, instead of directly accessing the SWIM or SCC controller chips. The IOP Manager provides sufficient functionality such that code outside of the IOP Manager should never need to directly access the IOPs.

IOP Numbering

The IOP Manager data structures are designed to handle multiple IOPs, and an upper limit of eight IOPs per system has been set. IOPs are numbered from zero to seven, with zero and one assigned to the SCC and SWIM IOPs respectively. The remaining IOP numbers are available for future expansion, and the possibility of IOPs on add on NuBus cards. The following constants are defined for referring to the IOPs.

```
CONST
   SccIopNum          = 0;          {SCC IOP is number 0}
   SwimIopNum         = 1;          {SWIM IOP is number 1}
   MaxIopNum          = 7;          {8 IOPs supported, numbered 0..7}
   NumberOfIOPs       = MaxIopNum+1;
```

Accessing IOP Memory Data

A routine is provided to copy data to and from a IOP memory, or compare HOST memory with IOP memory. Additionally it provides a special mode to allow a series of patches to be applied to IOP memory as an atomic operation (during which time, all other processing on the main CPU and on the IOP being patched will cease), in which case `imHostAddr` is passed a pointer to a series of packets, terminated by a zero length packet, that have the same format as the contents of an initial IOP code image resource, and the `imByteCount` and `imIopAddr` fields are ignored. To copy IOP data, the `_IOPMoveData` trap should be called with parameters passed as follows.

```
CONST
                                         {imCopyKind encodings}
   imIopToHost           = 0;            {Copy from IOP to main CPU}
   imHostToIop           = 1;            {Copy from main CPU to IOP}
   imCompare             = 2;            {Compare Host memory with IOP memory}
   imPatchIop            = 3;            {Patch IOP memory}

TYPE IOPMoveInfo =
   RECORD
      imCopyKind:        SignedByte;     {kind/direction of copy}
      imIOPNumber:       SignedByte;     {IOP Number (0..MaxIopNum)}
      imByteCount:       Integer;        {Count of bytes to copy (except PatchIop)}
      imHostAddr:        Ptr;            {Host CPU memory address}
      imIopAddr:         Integer;        {IOP memory address (except PatchIop)}
      imCompRel:         SignedByte;     {-1 if IOP < HOST, 0 if IOP = HOST,
                                           1 if IOP > HOST }
      imReserved:        SignedByte;     {unused, Reserved}
   END;
```

Trap Name: `_IOPMoveData` ($A088)

On Entry:

A0 Address of `IOPMoveInfo` record

On Exit:

D0 result code as follows

`NoErr`	Operation was successful.
`ParamErr`	Operation was unsuccessful for one of the following reasons. `imCopyKind` is out of range. `imIOPNumber` is out of range. The specified IOP does not exist, or is not initialized.

Message Passing in General

The IOP Manager provides an asynchronous message passing mechanism to allow tasks on the Host and IOP processors to communicate with each other. Message transactions may be initiated by either the Host or the IOP processor. There can be as many as seven message transactions occurring concurrently in each direction (for a total of 14 messages per IOP). The message data can be up to 32 bytes in length, and its contents is not dictated by the IOP manager. The message format, as well as the message number, is agreed upon by the sender and receiver of the message.

A message transaction actually consists of four phases. In the first phase, the transmitting processor copies the Message into a shared memory message buffer, and notifies the receiving processor that a message has been sent. In the second phase, the receiving processor will read the message and take whatever action is necessary to process it. In the third phase, the receiving processor will write a reply message into then shared memory message buffer, and then notify the transmitting processor that the message processing is complete. Finally in the fourth phase, the transmitting processor will copy the reply from the shared memory message buffer, and indicate that the transaction is complete, and call the completion handler associated with sender of the message.

The `_IOPMsgRequest` trap is provided to handle all message passing operations.

```
CONST
   MaxIopMsgNum       = 7;      {Message numbers range 1..7}
   MaxIopMsgLen       = 32;     {Message length range is 0..32 bytes}

                                {irRequestKind encodings}
   irSendXmtMessage   = 0;      {Send Transmit msg, Read reply when done}
   irSendRcvReply     = 1;      {Send Receive reply, Wait for next Receive Msg}
   irWaitRcvMessage   = 2;      {wait for Receive Message}
   irRemoveRcvWaiter  = 3;      {remove wait for receive message request}

TYPE IOPRequestInfo =
   RECORD
      irQLink:        Ptr;        {pointer to next queue element}
      irQType:        Integer;    {Queue Element type}
      irIOPNumber:    SignedByte; {IOP Number (0..MaxIopNum)}
      irRequestKind:  SignedByte; {kind of message request to perform}
      irMsgNumber:    SignedByte; {Message number (0..MaxIopMsgNum)}
      irMessageLen:   SignedByte; {Message Buffer length (0..MaxIopMsgLen)}
      irReplyLen:     SignedByte; {Reply Buffer length (0..MaxIopMsgLen)}
      irReqActive:    SignedByte; {$FF when active, or queued, $00 when completed}
      irMessagePtr:   Ptr;        {Message buffer address}
      irReplyPtr:     Ptr;        {Reply buffer address}
      irHandler:      ProcPtr;    {Completion handler procedure address}
   END;
```

Trap Name: `_IOPMsgRequest` ($A087)

On Entry:

A0 Address of `IOPRequestInfo` record

On Exit:

D0 result code as follows

`NoErr`	Operation was successful.
`ParamErr`	Operation was unsuccessful for one of the following reasons. `irRequestKind` is out of range.

`irIOPNumber` is out of range.
The specified IOP does not exist, or is not initialized.
`irMsgNumber` is out of range, or not supported by this IOP.
`irMessageLen` is out of range.
`irReplyLen` is out of range.
`irSendRcvReply` when another receiver already exists.
`irSendRcvReply` while request is still active.
`irWaitRcvMessage` when another receiver already exists.
`irRemoveRcvWaiter` when no receive waiter exists.

Host Initiated Message Passing Transactions

To send a message from the Host to the IOP, the `_IOPMsgRequest` trap is used with the `IOPRequestInfo` parameter block setup as follows.

`irRequestKind` should be set to `irSendXmtMessage`.

`irIOPNumber` and `irMsgNumber` should be set to indicate the receiver of the message.

`irMessagePtr` should point to the message data to be sent, and `irMessageLen` should indicate the length of the message data (which may be zero in the unlikely case when no message data is needed, in which case `irMessagePtr` is not used).

`irReplyPtr` should point to the buffer to receive the reply from the IOP, and `irReplyLen` should indicate the length of the reply buffer (which may be zero if no reply is expected, in which case `irReplyPtr` is not used).

`irHandler` should point to the procedure to be called when the IOP replies to the message, or should be set to zero if no handler is needed.

When the `_IOPMsgRequest` is made with the parameters described above, the following operations occur. The parameters are checked for validity, returning with an error if invalid. The request is now marked as being active (`irReqActive` will be set to `$FF`). The new request will be placed at the end of the transmit message queue associated with this message. If there was already an active request in progress for this message (the new request is not at the head of the queue), it will return now, with `NoErr`, and the request will be processed when the request ahead of it completes.

The first phase of the message transaction now begins, by copying `irMessageLen` bytes of message data pointed to by `irMessagePtr`, into the shared memory message buffer in IOP memory. A message state byte is also updated to indicate that a new message has been sent, and the Host processor will interrupt the IOP to notify it that a message needs processing. This trap will now return with `NoErr` indicating that the request processing has started.

The IOP will now go through the second and third phases of the message transaction, while the Host processor is free to proceed with other processing. At the end of the third phase, the IOP will interrupt the Host to notify it that a transmit message has completed.

The Host interrupt handler will process the fourth phase of the transaction, by searching the message states to find which message has completed, and then locate and de-queue the `IOPRequestInfo` parameter block at the head of the queue for that message. It will copy irReplyLen bytes of the reply from the shared memory message buffer in IOP memory to the buffer pointed to by irReplyPtr. The message state byte is also updated to indicate that a reply has been received. The completed request will be added to a queue of completed requests that will be processed by a deferred task which will run after all other interrupt processing has completed.

If there were any other message requests queued up for this message, the request at the head of the queue will be initiated now.

Finally the deferred task will run, still at interrupt stack level, but with interrupt priority lowered to zero, so all the rules about causing heap moves, and use of unlocked handles, etc. still apply. It will go through each entry in its completion queue, de-queuing it from the queue, setting irReqActive to $00 to indicate that the request is complete, and will call the procedure pointed to by irHandler, unless the pointer is zero. Register A0 will point to the IOPRequestInfo parameter block associated with this request, which will allow the handler to find its private storage (by appending or prepending additional data to the parameter block, or by embedding the parameter block within the private storage, and address other data relative to the parameter block). The handler, like most other Macintosh interrupt handlers, may destroy registers A0-A3, and D0-D3, and must preserve all other state.

Due to the asynchronous nature of this call, care should be taken to be sure that the pointers irMessagePtr, irReplyPtr, irHandler, and the IOPRequestInfo record itself, point to memory that will not be relocated, and are not allocated on a portion of the stack that might be de-allocated during the asynchronous processing of the request.

IOP Initiated Message Passing Transactions

For the Host to receive a message from the IOP, the _IOPMsgRequest trap is used in one of three ways. Two of these are required, and the third is optional and may be needed in some situations. The IOPRequestInfo parameter block setup as follows.

For the first required call, irRequestKind should be set to irWaitRcvMessage. This call is used to install a handler to receive messages initiated by the IOP.

irIOPNumber and irMsgNumber should be set to indicate the sender of the message.

irMessagePtr should point to the message buffer to receive the message, and irMessageLen should indicate the length of the message buffer (which may be zero in the unlikely case when no message data is needed, in which case irMessagePtr is not used).

irHandler should point to the procedure to be called when the IOP sends the message, or should be set to zero if no handler is needed.

When the _IOPMsgRequest is made with the parameters described above, the following operations occur. The parameters are checked for validity, returning with an error if invalid. If there was already a handler installed to receive this message, ParamErr is returned. Otherwise, the request is now marked as being active (irReqActive will be set to $FF). The request will be installed as the receive message handler associated with this message, and it will return now, with NoErr.

When the IOP wants to send a message, it will initiate the first phase of the message transaction by copying the message into the shared memory message buffer in IOP memory. The message state byte is also updated to indicate that a new message has been sent, and the IOP processor will interrupt the Host to notify it that a message needs processing.

The Host processor will now go through the second phase of the message transaction, while the IOP is free to proceed with other processing. The Host interrupt handler will process the second phase of the transaction, by searching the message states to find which message has completed, and then locating the `IOPRequestInfo` parameter block associated with that message. It will copy `irMessageLen` bytes of the message from the shared memory message buffer in IOP memory to the buffer pointed to by `irMessagePtr`. The message state byte is also updated to indicate that a message has been received. The request will be added to a queue of requests that will be processed by a deferred task which will run after all other interrupt processing has completed.

Finally the deferred task will run, still at interrupt stack level, but with interrupt priority lowered to zero, so all the rules about causing heap moves, and use of unlocked handles, etc. still apply. It will go through each entry in its queue, de-queuing it from the queue, setting `irReqActive` to `$00` to indicate that the request is complete, and will call the procedure pointed to by `irHandler`, unless the pointer is zero. Register `A0` will point to the `IOPRequestInfo` parameter block associated with this request, which will allow the handler to find its private storage (by appending or prepending additional data to the parameter block, or by embedding the parameter block within the private storage, and address other data relative to the parameter block). The handler, like most other Macintosh interrupt handlers, may destroy registers `A0-A3`, and `D0-D3`, and must preserve all other state.

Due to the asynchronous nature of this call, care should be taken to be sure that the pointers `irMessagePtr`, `irReplyPtr`, `irHandler`, and the `IOPRequestInfo` record itself, point to memory that will not be relocated, and are not allocated on a portion of the stack that might be de-allocated during the asynchronous processing of the request.

The second required call is used after the `irHandler`, procedure has been called, to send a reply message back to the IOP, completing the message transaction.

The same `IOPRequestInfo` parameter block that was used to receive the message must be used to send the reply, with `irRequestKind` modified to be `irSendRcvReply`. If the request is still marked as being active (ie. waiting for the IOP message), `ParamErr` will be returned.

`irReplyPtr` should point to the buffer to send as the reply to the IOP, and `irReplyLen` should indicate the length of the reply buffer (which may be zero if no reply data is needed, in which case `irReplyPtr` is not used).

The Host processor will now enter the third phase of the message transaction, by copying the reply message into the shared memory message buffer, and updating the message state to indicate that the message is completed. The Host will now interrupt the IOP to notify it that a message has completed.

The Host processor will now again prepare to receive messages initiated by the IOP, by repeating the same operations as above when `irRequestKind` was set to `irWaitRcvMessage`.

The third call, which, depending upon the situation, may or may not be needed, is used to remove a receive message handler.

The same `IOPRequestInfo` parameter block that was used to wait for the message must be used to remove the handler, with `irRequestKind` modified to be `irRemoveRcvWaiter`. otherwise

ParamErr will be returned. If the request will marked as being complete, and there will no longer be a handler associated with that message.

IOP Installation and Removal

The IOP Manager provides an interface that allows IOPs to be dynamically added and removed. This routine would most likely be used to install an IOP that resides on some sort of expansion card, although it is called internally by the IOP Manager initialization routine at system startup time to install the two currently defined IOPs. The following data structures and calls are used to install, remove, and access the information about an IOP.

```
TYPE IOPMsgEntry =
   RECORD
      RcvMsgInfoPtr:    Ptr;          {Ptr to rcv msg handler info (IOPRequestInfo)}
      Reserved:         Word;         {unused, reserved}
      XmtMsgQHdr:       QHdr;         {queue of transmit requests (IOPRequestInfo)}
   END;

TYPE IOPInfo =
   RECORD
      IopAddrRegPtr:    Ptr;          {Ptr to IOP RAM Address Reg (word)}
      IopDataRegPtr:    Ptr;          {Ptr to IOP RAM Data Reg (byte)}
      IopCtlRegPtr:     Ptr;          {Ptr to IOP Control Reg (byte)}
      BypassHandler:    ProcPtr       {Bypass Mode Interrupt Handler Address}
      MaxXmt:           SignedByte;   {Highest Transmit message number}
      MaxRcv:           SignedByte;   {Highest Receive message number}
      Reserved:         Word;         {unused, Reserved}

      MoveReqInfo:      IOPRequestInfo;  {request info for rcv message 1}
      MoveReqBuffer:    IOPMoveInfo;  {msg/reply buffer for rcv message 1}

      MsgTable:         array [1..7] of IOPMsgEntry;     {message handler info}
   END;

CONST
   iaInstallIOP         = 0;
   iaGetIOPInfo         = 1;
   iaRemoveIOP          = 2;

TYPE IOPAccessInfo =
   RECORD
      iaAccessKind:     SignedByte;   {kind of request to perform}
      iaIOPNumber:      SignedByte;   {IOP Number (0..MaxIopNum)}
      Reserved:         Word;         {unused, Reserved}
      iaIOPInfoPtr:     Ptr;          {pointer to IOPInfoRecord}
   END;
```

Trap Name: _IOPInfoAccess ($A086)

On Entry:

A0 Address of `IOPAccessInfo` record

On Exit:

D0 result code as follows

`NoErr`	Operation was successful.
`ParamErr`	Operation was unsuccessful for one of the following reasons.
	`iaAccessKind` is out of range.
	`iaIOPNumber` is out of range.
	When `iaAccessKind` is `iaInstallIOP`, the following are possible
	`iaIOPNumber` already exists.
	The '`iopc`' resource for `iaIOPNumber` could not be found.
	`iaIOPNumber` failed its RAM test, or failed to initialize.

Associated with each installed IOP, there is an `IOPInfo` record. It contains the information needed by the IOP manager to process requests for that IOP. `IopAddrRegPtr` is a pointer to the 16 bit IOP RAM Address register, `IopDataRegPtr` is a pointer to the 8 bit IOP RAM Data register, and `IopCtlRegPtr` is a pointer to the 8 bit IOP control and status register. BypassHandler is a pointer to the interrupt handler to be called when the IOP interrupts the Host, and the IOP was in bypass mode. `MaxXmt` and `MaxRcv` are contain the highest message number supported by the IOP in each direction. `MoveReqInfo` and `MoveReqBuffer` are used to support the IOP requested data movements which will be described later. `MsgTable` is an array of seven `IOPMsgEntry` records associated with the seven possible message numbers that an IOP can support.

An `IOPMsgEntry` record contains the information used to process transmit and receive messages for a given message number. `RcvMsgInfoPtr` is a pointer to an `IOPRequestInfo` record which is the receive request associated with this message number. `XmtMsgQHdr` is a queue of `IOPRequestInfo` records, the head of the queue is the transmit message that is currently active for this message number.

The `IOPAccessInfo` record is used with the `_IOPInfoAccess` trap to access or change information associated with a given IOP. `iaAccessKind` describes the operation to perform, `iaIOPNumber` is the IOP to operate upon, and `iaIOPInfoPtr` is a pointer to an `IOPInfo` record that is either passed in, or returned, depending upon the operation.

When `iaAccessKind` set to `iaGetIOPInfo`, `iaIOPInfoPtr` will be returned with the pointer to the `IOPInfo`, or will be zero if there is no `IOPInfo` for that IOP.

When `iaAccessKind` set to `iaRemoveIOP`, the `IOPInfo` for the specified IOP will be removed, and a pointer to it will be returned in `iaIOPInfoPtr`.

To install a new IOP, `iaAccessKind` is set to `iaInstallIOP`, and `iaIOPInfoPtr` should point to the `IOPInfoRecord` for this IOP. The `IopAddrRegPtr`, `IopDataRegPtr`, `IopCtlRegPtr`, and `BypassHandler` pointers in the `IOPInfo` record must be setup prior to making the `_IOPInfoAccess` trap. The IOP will be installed, and initialized as follows.

Since a IOP doesn't have any ROM, it will execute code out of part of its 32K bytes of RAM. The main CPU ROM will contain the code for each IOP, usually stored as a ROM resource.

_RGetResource is used to locate the resource, so that is may be overridden by a resource in another resource file. If the resource is not found at all, paramErr will be returned. The ResType is 'iopc' and theID corresponds to the IOP number. The IOP Code resource will be organized as a sequence of variable length packets, terminated by a zero length packet. A packet will have a three byte header followed by 0..255 bytes of data. The first byte of the header contains the length of the packet data. The remaining 2 bytes of the header contains the IOP memory starting address where the data from this packet should be placed. When the code gets loaded into IOP memory, any bytes that were not loaded will be set to zero. This should allow a significant saving in the size of the IOP code resource, since long strings of zero bytes (3 or more), can be compressed out of the load image.

The Host CPU initializes the IOP hardware by first putting the IOP into the RESET state, which prevents it from executing instructions. It will then write a pattern of $FF to each byte in IOP memory. It will then read each byte of IOP memory, checking it to make sure that it was written as $FF, and replace it with a value of $00. Next it will check all of IOP memory to make sure that it was correctly set to zeros. Finally, the IOP code will be loaded, and read back to verify that it was loaded correctly, and the main CPU will put the IOP into the RUN state, to allow it to start executing the now loaded IOP code. The Host CPU will wait for the IOP to finish executing its initialization code, and then install the Move Message Request handler which will be described later. If any of these operations fail or timeout, it will be considered a fatal hardware failure for that IOP, and any future calls to the IOP Manager to communicate with the faulty IOP will return an error status indicating that that IOP does not exist.

Move Request Receive Message Handler

While there is no hardware support that would allow the IOP to access the main CPU memory, it is necessary for some IOP based drivers to do so. This is achieved by having the IOP send a message to the main CPU requesting it to copy the data to or from the IOP memory. IOP to Host receive message number one will be used on all IOPs to support this operation. The IOP Manager provides and installs a handler for this message for all of the IOPs in the system, to perform the data movement requests. The format of the message that the IOP sends is the same format as the IOPMoveInfo parameter block used by the _IOPMoveData trap, with the exception that the IOP does not fill in the imIOPNumber field. When the Host receives the message from the IOP, it will fill in the imIOPNumber field with the correct IOP number, and call the _IOPMoveData routine to process the move request. When the move completes, it will send the IOPMoveInfo parameter block back to the IOP as the reply message, and wait for the next move request. There are currently no assigned functions for Host to IOP transmit message number one.

Initialization

Early in the booting process (before any drivers are opened) the StartManager will call the IOP manager routine InitIOPMgr which will create the IOP Manager global data structures in the system heap, and make _IOPInfoAccess calls to install and initialize the IOP hardware for the two currently defined IOPs. Additionally, it will start up a VBL task that will monitor the IOPs every few seconds to see that they are still alive. This is a debugging aid, and will not be part of the final production version of the IOP Manager.

The globals data structures used by the IOP Manager are as follows. Note that IOPmgrVars does not point to the beginning of the record, instead it points to the IOPInfoPtrs field of the record, so that this data structure may expand in both directions in the future.

```
CONST
   IOPmgrVars          = $0C28;       {low mem Ptr to IOPInfoPtrs field of
                                       IOPMgrGlobals record}
TYPE IOPMgrGlobals =
   RECORD
      Filler1:         Word;          {force nice long word alignments}
      VTask:           VBLTask;       {IOP polling task}
      DTask:           DeferredTask;  {completion caller deferred task}
      Filler2:         SignedByte;    {unused, reserved}
      DTaskQueued:     SignedByte;    {-1 if task queued or running, 0 when done}
      CompleteQHdr:    QHdr;          {queue of completed requests}
      IntHandlerPtr:   ProcPtr;       {pointer to IOP Interrupt handler}

      {IOPmgrVars points here}
      IOPInfoPtrs:     Array [0..MaxIopNum] of Ptr;
                                      {Ptrs to IOPInfo records for each IOP}
      SCCIOPInfo:      IOPInfo;       {info for IOP 0 (SCC)}
      SWIMIOPInfo:     IOPInfo;       {info for IOP 1 (SWIM)}
   END;
```

Message State Details

A message can be in one of four states at any given time. The message states, and the transition order is as follows. This protocol will work in a shared memory environment without the need for indivisible memory operations, as long as there is a single sender and a single receiver processor associated with each message.

- *Idle* – This state indicates that the message contents are invalid, and no action is needed by the sending or receiving processor. This states indicates to the sending processor that this message is available for use. This state is only set by the sending processor.

- *New Message Sent* – This state indicates to the receiving processor that the message data is now valid, and that it may now receive it, and start processing it. The sending processor also interrupts the receiving processor to indicate that there is a message that is now in this state. The receiving processor now owns the message data, and the sending processor can no longer modify it. This state is only set by the sending processor.

- *Message Received* – When the receiving processor has received a message, and has started processing it, the message state changes to this state, to acknowledge the the interrupt, and to indicate that it has started processing the request. This state is only set by the receiving processor.

- *Message Completed* – This state indicates to the sending processor that the receiving processor has completed processing the message, and any returned message data is now valid. The receiving processor also interrupts the sending processor to indicate that there is a message that is now in this state. The sending processor now owns the message data, and the receiving processor can no longer modify it. This state is only set by the receiving processor.

- *Idle* – When the sending processor has acknowledged the message completed interrupt for the message, and processed any message data that was passed back by the receiving processor, it changes the message state to *Idle* to release the message. This state is only set by the sending processor.

While the desired order of state transitions is the order listed above, it may be desirable for some driver implementations to skip some state transitions. It is allowable to transition directly from *New Message Sent* to *Message Completed* without going through *Message Received*. It is also allowable to transition directly from *Message Completed* to *New Message Sent* without going through *Idle*.

The message state will occupy 1 byte for each message, the encodings are as follows.

```
CONST
   MsgIdle            = 0;      {message buffer idle}
   NewMsgSent         = 1;      {new message just sent}
   MsgReceived        = 2;      {message received, and being processed}
   MsgCompleted       = 3;      {message processing complete, reply available}
```

Shared Memory Data Structures

The 65CX02 organizes its memory into pages of 256 bytes. Page zero is used by special addressing modes, and page one is used for the 65CX02 stack. We will allocate a total of two pages (pages 2 and 3) of shared memory for message passing, where the first page (page 2) will be used for messages sent from the main CPU to the IOP, and the second page (page 3) will be used for messages sent from the IOP to the main CPU. The layout of each of these pages will be as follows.

Offset	Length	Description
$00	1	Max Message Number – Highest message number used by this IOP.
$01	1	Message State for message 1
$02	1	Message State for message 2
...	...	...
$06	1	Message State for message 6
$07	1	Message State for message 7
$08	24	Reserved
$20	32	Message Data for message 1
$40	32	Message Data for message 2
...	...	...
$C0	32	Message Data for message 6
$E0	32	Message Data for message 7

In addition to the message passing, there are two special bytes in IOP memory (IOP addresses $021F and $031F) which are used as follows.

IOP memory address $021F will be used to synchronize patching of code in a running IOP. When a IOP is otherwise idle, it will check this byte, and if it contains the value *New Message Sent* , it will disable all interrupts, and change the value of this byte to *Message Completed*, and then wait in a loop (with interrupts still disabled) waiting for the value in this byte to change to *Idle*. It will finally re-enable interrupts, and continue with its idle loop. When the IOP Manager wants to patch code in a running IOP, it will disable all interrupts, and store the value *New Message Sent* into this patching flag byte in IOP memory, then it will poll the byte, waiting for the value to change, at which time it will know that the IOP is in a state ready to be patched. The IOP Manager will then apply the patches, and finally store the value *Idle* into the patching flag byte, and re-enable interrupts.

IOP memory address $031F will be used to allow the main CPU to check to see if a IOP is still alive (it may have hung or crashed, due to hardware or software problems). The IOPs will store $FF into this alive flag byte when it is otherwise idle. The VBL task in the IOP manager will read this byte, and clear it, every few seconds to make sure that the IOPs are still alive

Interrupts

When the Host processor detects an interrupt from an IOP, it needs to call the IOP Manager's interrupt handler routine, and pass the interrupting IOPs number in the low word of register `D0`, and flags indicating which messages completion routine must run immediately, and which ones can be deferred in the upper 2 bytes. The highest byte refers to Host to IOP message completions, and the next byte refers to IOP to Host message requests. Each bit of the byte is associated with the corresponding message number, and a zero bit indicates that the completion routine can be deferred, while a one indicates that it must be run immediately. This allows message processing to proceed in an environment (like the keyboard polling code for MacsBug) where interrupts are disabled (which prevents deferred tasks from running), and the IOP interrupts are polled, and the interrupt handler is called to service the request, without ever enabling interrupts. The address of the interrupt handler may be found in field `IntHandlerPtr` of `IOPMgrGlobals`.

Since each IOP can cause two different kinds of interrupts on the main CPU, we will use one interrupt (INT0) to indicate that there is a *Message Completed* in the main CPU to IOP message page, and the other interrupt (INT1) will be used to indicate that there is a *New Message Sent* in the IOP to main CPU message page. If neither interrupt flag was set, and the IOP is in Bypass mode, the handler pointed to by field `BypassHandler` of the `IOPInfo` record for the interrupting IOP will be called, and the interrupting IOPs number will still be in register `D0`. The interrupt handler may destroy registers `A0-A3`, and `D0-D3`, and must preserve all other state.

Implementation

The IOP Manager is being implemented in MPW 68020 assembly code (using newer addressing modes and opcodes where appropriate). It is expected to use less than 1.5K bytes of ROM space, and less than 500 bytes of RAM space in the system heap to support the two currently defined IOPs.