Development of 360/370 Architecture - A Plain Man's View

February 10th, 1989

P.J. Gribbin (Gvn/PJG)

EDS
c/o Rolls-Royce Plc
P.O. Box 31
Derby
England, DE24 8BJ
Tel:- +44-1332-522016

Jeff.Gribbin@EDS.Com

1.0Abstract

Since the introduction of System /360 in the 1960's, the architecture has evolved through a few major steps and several minor steps. The author has had occasion to need a "plain man's" appreciation of the architecture and its evolution on a number of occasions in order to facilitate problem determination and configuration planning. This presentation is a high-level introduction to the main features of /360 architecture, followed by an equally high-level look at how that architecture evolved through /370 into /370-XA. It is aimed at "beginning" systems programmers who have an interest in the architecture - particularly VM systems programmers although others may benefit.

Note: Throughout this document, the pronouns "he" and "his" are used in a manner that is intended to pertain to any sentient organism that qualifies for the role being described. They are *not* intended to suggest any limitation on the sexual orientation (if any) of the physical body the entity being alluded to may currently happen to inhabit.

2.0Preface

Today, the strategic IBM mainframe environment is delivered via the IBM Enterprise Systems Architecture/370[1] This architecture is the latest in a series of steps that began in the mid-1960's with the introduction of the IBM System/360 architecture. There is little reason to believe that ESA/370 is the final step in this evolution.

As a VM Systems Programmer since 1974/75 (the earliest VM system I recall using was VM/370 Release 2 PLC 8), I have been concerned with IBM System /370 architecture for many years. After all, it's not only the architecture that VM/370 runs on, it's also the application interface that VM presents to my users.

My interest in /370 architecture has led me to look into its roots (in System /360) and into its successors (/370-XA and ESA/370). From this "plain man's" understanding of the architectures I have put together this presentation which is intended to walk through the 20-odd years of evolution. On the way I shall point out the features that I find of interest and indicate what I believe were the various constraints that had to be relieved at each evolutionary step.

This tour of the architecture is not intended in any way to be a definitive study. You should check any statement made in this presentation with the appropriate original sources before relying on its accuracy.

Any opinions or attitudes expressed in this presentation are entirely my own; they should not be construed as being in any way the formal opinions or attitudes (if any) of Rolls-Royce plc on the matter in hand.

I am indebted to Lee and Melinda Varian, and Serge Goldstein (all of SHARE installation code PU), who took the time to read the draft versions of this presentation and straighten me out whenever my understanding differed from reality. Their contribution is greatly appreciated.

[1] Trademark of IBM Corporation.

3.0Table of Contents

4.0List of Illustrations

5.0In the beginning ?

During the 1950's, Data Processing came of age. There were already plenty of Data Processing machines in existence - sorters, collators, tabulators and so on - but nobody had seriously considered "programming" the scientists' new toy - the computer - to perform these simple and repetitive tasks. "Computers" were devoted almost entirely to the processing of computationally intensive tasks, bringing substantial amounts of processing power to bear on relatively small amounts of data. A program would often run for several hours on a diet of just a few hundred cards, and then only regurgitate a few thousand lines of print.

However, once the idea had germinated, demand for computers as data processing machines boomed and new machines, such as the IBM 1401, were built to meet this demand.

Experience with these early "Data Processors" demonstrated that they were radically different beasts when compared to their scientifically-orientated predecessors. They had to handle large volumes of data with relatively little "computing" being required on each individual item; they needed a simple instruction set so as to keep the time spent repetitively extracting and decoding instructions as low as possible, and they needed to deliver a near-absolute guarantee of data integrity. I/O errors were only forgivable if they were detected, and everything must be checked. Card punches must be designed to read back and verify the cards they were punching, and so on.

Building on their experience with the 1401, IBM decided to implement a wholly new architecture specifically designed both for data processing and to be compatible across a wide range of performance levels. A key factor in this design was its ability to *move data.* I don't possess the necessary figures to be certain, but I would not be surprised to learn that even the earliest /360's could move data in and out of main storage at a speed comparable to that achieved by today's workstations. A modern 3090, with 128 channels, can easily cope with data rates well in excess of 200MB per second. (Yes, that's a complete 3380-E volume every six seconds!)

These new machines were designed to handle effectively the enormous disparity in speed between the peripherals and the processor. The difference is two or three orders of magnitude; easily-managed overlapping of I/O and Central Processing, along with the ability to allow concurrent "multi-programming" of independent units of work, is built into the system.

The System /360 design was also intended to be a design for the future. Huge amounts of extra capacity were built into the architecture, such as the ability to address up to 16 Megabytes of main storage, and allowing for as many as seven data channels.

The data-moving capability was catered for by delegating practically all the I/O management to independent co-processors called *channels.* These processors could execute the data-moving programs (called Channel Programs, made up of Channel Command Words, or CCW's) concurrently with *each other and the central processor.* When the channels required the CPU's attention, they could signal the processor by generating an interrupt. This very simple switching mechanism could take control away from an application when the processor was required to deal with some more urgent task in such a way that control could be returned later to the interrupted application without the application even needing to be aware that it had been interrupted.

We shall now look at the essential components of a System /360 machine.

6.0 A Child is Born ?

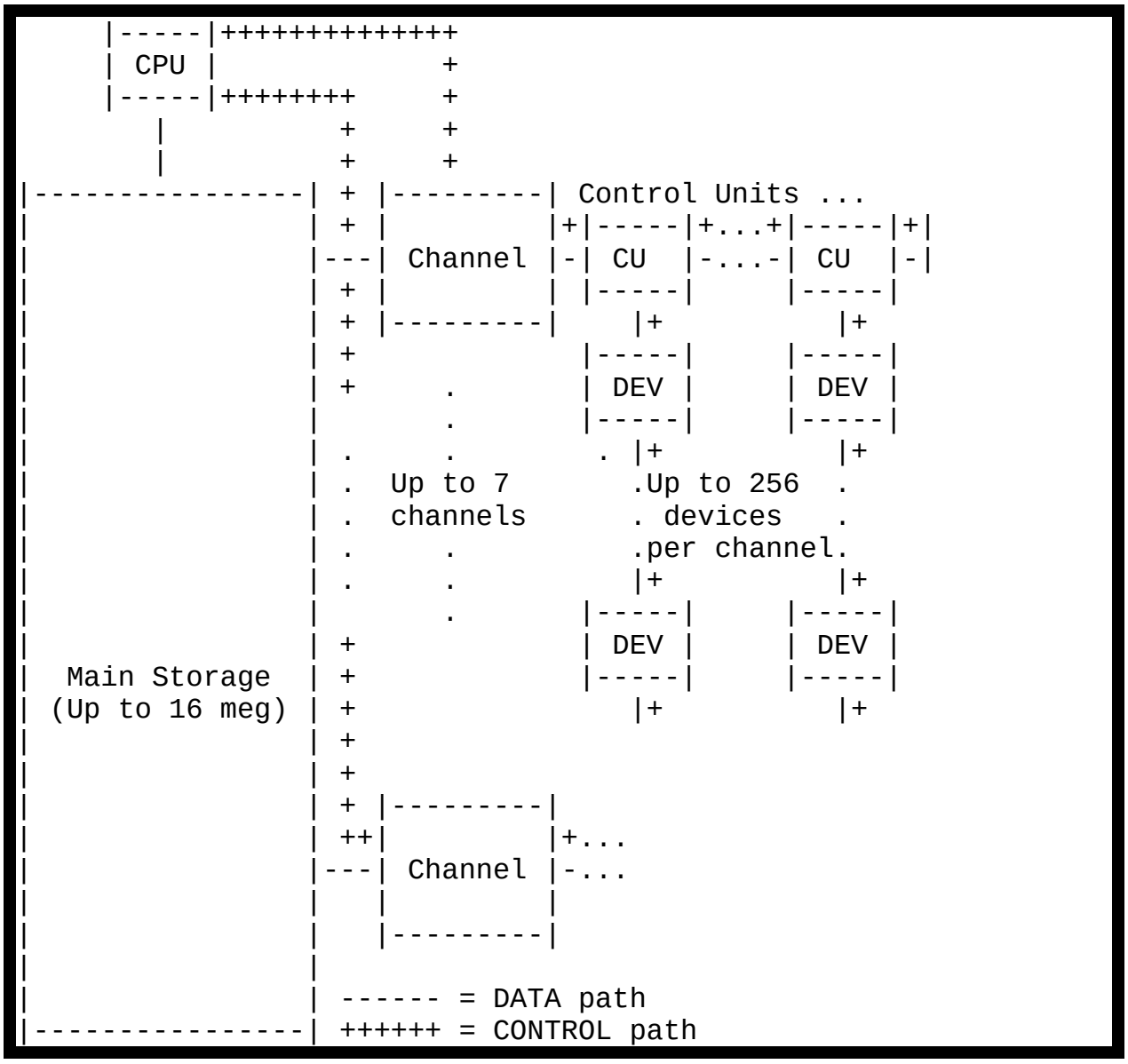


Figure 1: Basic Components of a /360

A System /360 machine contains five major components:-

1. One Main Storage, maximum size 16 megabytes.
2. One or two Central Processors.
3. One or more Channels, up to a maximum of 7.
4. Control Units, which connect to Channels.
5. Devices, which connect to Control Units.

Let's look a little more closely at each component in turn.

6.1Main Storage

Known colloquially as "core" after an early implementation which used miniature rings (or "cores") of ferrite for each bit of memory, Main Storage is an array of data locations, each location containing eight *bi*nary digi*ts*, or "*bits*". Each 8-bit location is known as a *byte*, and has its own unique numerical address, starting from zero and rising consecutively to the last byte of Main Storage. The following structure is imposed on Main Storage:-

- The byte at any address that is divisible by eight, along with the bytes at the next seven consecutive storage addresses, is known as a *doubleword*.
- The byte at any address that is divisible by four, along with the bytes at the next three consecutive storage addresses, is known as a *fullword*.
- The byte at any address that is divisible by two, along with the byte at the next consecutive storage address, is known as a *halfword*.

(Inside VM-land, we now also use the term "quadword". This refers to a byte at any address that is divisible by sixteen, along with the bytes at the next fifteen consecutive storage addresses. Real-device blocks in DMKRIO are an integral number of "quadwords" in size. However, "quadwords" have no formal place in the architecture, they're merely a VM-ism.)

Programs, of course, place their own "soft" structure on Main Storage for the duration of their time in storage. These structures can be of any size (within the limits of available storage) and any alignment. Data fields referred to in machine instructions always occupy consecutive storage addresses and are referenced by the Central Processor via the start address (that is the address of the lowest-addressed byte of the field) and the length. Sometimes the length is explicitly specified in the program, sometimes it's implicit in the instruction.

Conventionally, Main Storage can be considered to be a ribbon, one byte wide, with location zero on the extreme left and the other locations arranged in ascending order to the right. For this reason, the start address of a field is often referred to as the left-hand end of the field.

Several fields in Main Storage are pre-defined by the architecture. These fields are used as communication areas between the software and the CPU and Channels. All these fields have addresses below 4096 which, because of the way the CPU is designed to allow programs to address Main Storage, means that they are directly accessible from programs running anywhere in the machine.

Although not original with the /360, an important feature of Main Storage is that it is shared indiscriminately by programs and data. Programs are read into storage and manipulated using exactly the same instructions that are used to manipulate data. It is easy for a program to dynamically modify itself as it runs - this may lead to code that is difficult to support and maintain, but it was a valuable feature in the storage-hungry 1960's. As far as the CPU is concerned, the only "program storage" is the instruction pointed to by the current PSW[2].

In order to protect the supervisor programs from damage by the application programs, Main Storage is divided into 2048-byte blocks, and each block can be assigned a *protection key.* A program is only allowed to modify data that lies inside blocks of Main Storage that have the same key as the current execution key. (The current key is held inside the CPU - see below.) The protection keys are assigned and changed dynamically by the supervisor software in charge of the machine, using privileged instructions.

Main Storage is designed to be accessed simultaneously by several "processors". Normally, these "processors" are one CPU and a number of channels, but there is nothing to prevent the configuration of more than one CPU onto a single Main Storage, and mechanisms were provided in the architecture to allow for this possibility. However, multiprocessing as a normal, commercial, "way of life" didn't really come about until later, so I have deferred consideration of multiprocessing until we get to System /370[3].

6.2The Central Processor

As we proceed on our tour, we shall realize that the Central Processor is unfortunately named. The central component of the system is, without doubt, Main Storage. However, Central Processing Unit (or CPU) is the name it was given, and it's the name we're now stuck with.

The CPU is responsible for extracting, decoding and executing machine instructions resident in Main Storage, initiating channel programs, and receiving and processing interrupts. It uses special storage locations wholly contained inside the CPU (called *registers*) along with predefined Main Storage locations below address 4096 to co-ordinate this activity. The CPU contains no built-in "programming" of its own other than the very basic "IPL" logic which is used to boot the software following a system reset. The CPU always runs under control of software which is resident in Main Storage; either privileged software (which is allowed to execute any instruction in the CPU's repertoire), or problem-state software (which is restricted to a subset of the available instructions).

Because the set of instructions that the CPU will execute is divided into two subsets, privileged and non-privileged, it is quite straightforward for the Supervisor software to protect itself from accidental or malicious damage caused by application programs. Applications are only given control of the CPU in Problem State and are thereby denied access to the instructions that affect the system as a whole. When an application needs a system service (such as maybe the execution

---

[2] Supporting this approach to Main Storage has cost an enormous price over the years as CPUs have been made to run faster and faster, and IBM has finally withdrawn this support in ESA/370. In ESA, a "serialization and checkpoint-synchronization function" has to be executed between modifying Main Storage and using that Main Storage as a target for the PSW.

[3] For the history buffs, the earliest operating system support for /360MP that I am aware of was OS/MVT support of a 360/65MP.

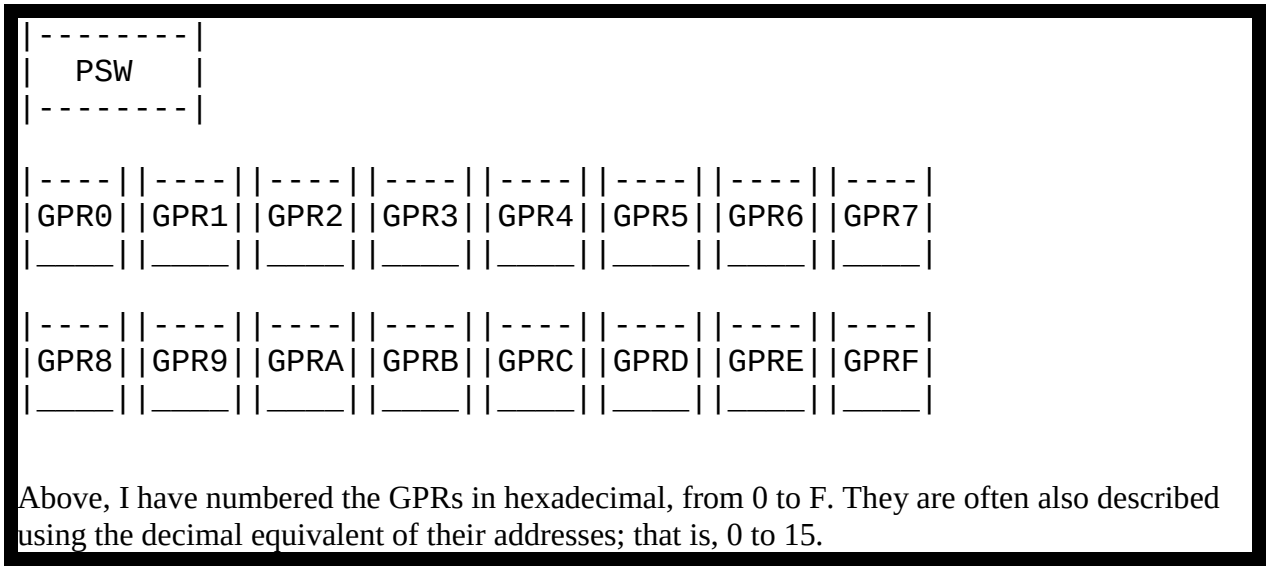
of an I/O operation) it asks the Supervisor (usually via a Supervisor-Call interrupt) to perform the service on its behalf. Prior to performing the service the Supervisor can analyze the request and ensure that it's acceptable.

The following registers are contained in a System /360 CPU. (Note: This is not a complete description.)

- The *Program Status Word* (or PSW) contains all the major control fields that the CPU requires in order to decide "what to do next". It defines which channels are allowed to interrupt the CPU, the current execution key for storage protection, the condition code, whether or not supervisor-state instructions are allowed, and the Main Storage address of the next instruction to be extracted, decoded and executed. Despite being called the Program Status *Word*, the PSW is actually eight bytes in size, like a *doubleword*.

- The *General Purpose Registers* (or GPRs) are sixteen special storage locations *wholly contained inside the CPU*, each of which is four bytes in size (the same size as a fullword). The GPRs can be loaded, modified and stored by the application programs, and are also used by the CPU as pointers to Main Storage.

Many of the System /360 CPU's instructions reference both an area in Main Storage and one or more GPR's. For these instructions it is usual to find that the Main Storage operand has to be properly aligned on its integral boundary. (That is, an 8-byte field must have an address exactly divisible by 8, a 4-byte field's address must be divisible by 4, and so on.) This proved to be of sufficient nuisance that later /360 models (and, of course, /370 models) lifted this restriction for most situations.



Above, I have numbered the GPRs in hexadecimal, from 0 to F. They are often also described using the decimal equivalent of their addresses; that is, 0 to 15.

Figure 2: Registers in a /360 CPU

6.3Channels

Channels are separate I/O processors that run independently of the CPU under control of Channel Programs that reside in Main Storage. System /360 architecture allows us to have up to

seven channels, each channel can be either a Multiplexor Channel or a Selector Channel. A multiplexor channel has the capability to be "multiprogrammed", and can overlap the execution of several "simultaneous" channel programs; a selector channel can only execute one channel program at a time. Generally speaking, multiplexors are intended for use with devices where the data transfer rate is slow when compared with the capacity of the channel as a whole (for instance, card readers and typewriter terminals), and selectors are intended for use with devices such as magnetic tapes whose data transfer capability more closely matches that of the channel.

A channel program is started by signals from the CPU. When the CPU interprets a request to initiate an I/O operation (SIO instruction), it passes the device address to the channel as part of the SIO, along with the contents of locations 72-75 (48-4B in hex) - known as the Channel Address Word (CAW). It knows which channel to signal because the first part of the device address is the channel-number (see Figure 4, on Page 12) The channel places the contents of the CAW in an internal register equivalent to the CPU's PSW. This register is known as the Unit Control Word, or UCW. A selector channel has only one UCW, a multiplexor channel has several. (Strictly, each UCW is part of a *subchannel* rather than a component in its own right. You can tell an IBM engineer's age by whether he says, "UCW", or "subchannel"!)

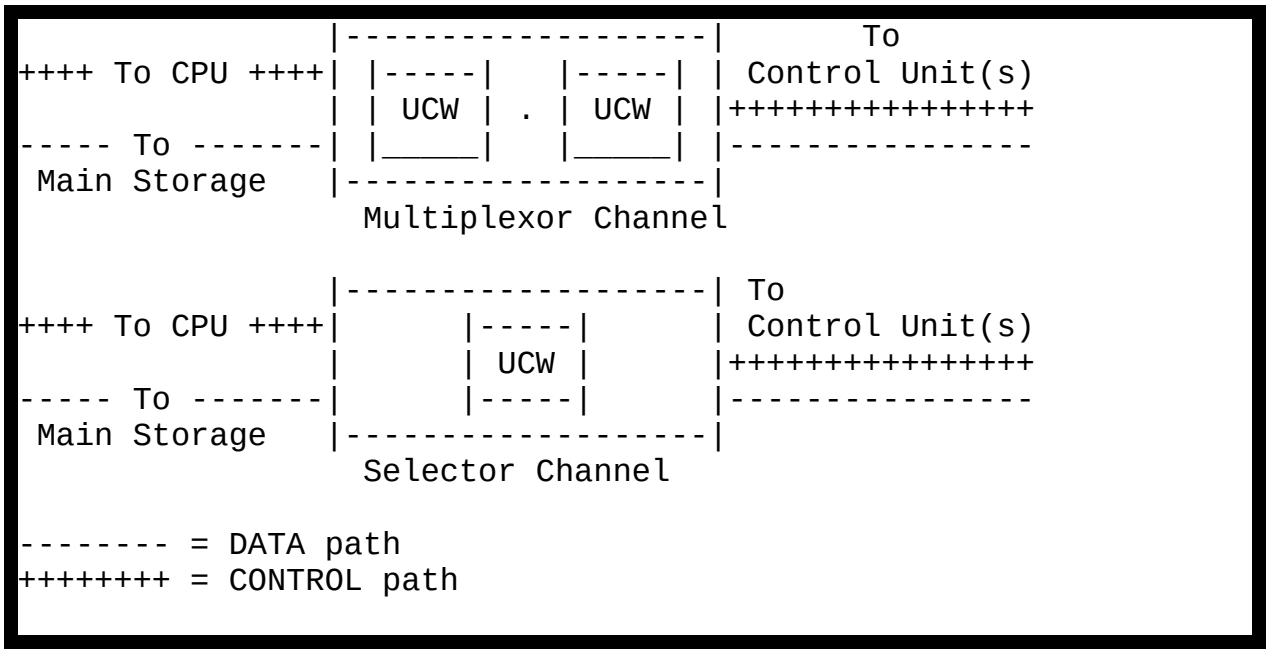


Figure 3: Multiplexor and Selector Channels

Having initialised its UCW (and assuming that there isn't any impediment to the channel program proceeding - such as device does not exist or is switched off), the channel then simply goes into its instruction fetch-and-execute cycle, just like any other processor, pulling CCW's out of Main Storage one at a time and obeying their commands. When the channel program completes normally (or possibly with errors), the channel signals an I/O interrupt to the CPU, passing back status information in the Channel Status Word (or CSW) at locations 64-71 (40-47 in hex). The CPU software is then responsible for decoding and acting upon the status in a relevant manner.

6.4Control Units

A control-unit's main task is to translate between the standardized "language" of the channel, and the highly esoteric "languages" of its devices. Control units are device-specific; you can't attach printers to tape controllers.

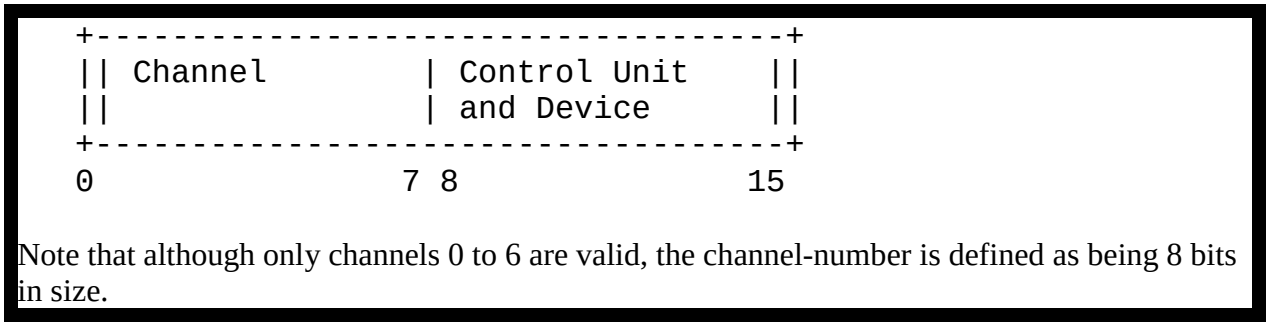
Sometimes control units are dedicated to serving a single device, such as the 2821 which serves a 1403 printer, and sometimes they are shared by many devices, such as the 2703 which can address up to 176 terminals. When several devices share a control unit, I/O for one device may have to wait while the control unit services I/O requests for other devices that it owns.

Our I/O management software needs to be aware of the presence of control units so that it can correctly respond to the status signals regarding control units sent back by the channels but, apart from this need, they're of diminishing interest compared with channels - which we can program - and devices - which hold our data.

6.5Devices

Devices attach to, and share, control units. Most of the status signals and requests exchanged between the channels and main storage - either in the CSW or as the result of the execution of channel programs - will relate to a single, specific, device.

Almost universally, application channel programs will only be interested in a device, and the data on that device. Any need to manage the channel or control unit as separate, visible, entities will be considered overhead. (Of course, nowadays, most applications even consider development of channel programs to be an overhead and use access methods provided by the Supervisor to convert their requests for data into channel programs.)



Note that although only channels 0 to 6 are valid, the channel-number is defined as being 8 bits in size.

Figure 4: Components of a System/360 Device Address

7.0Interrupts

Interrupts are such a fundamental part of any realistic approach to running a /360 cost-effectively that I feel it's worth looking a little more closely at how they work.

At System IPL time, the CPU is *disabled* for interrupts. That is, bits 0 to 5 (the Channel Masks), bit 7 (External), bit 13 (Machine Check) and bits 36 to 39 (Overflow and Significance) of the PSW are all set to zero.

During System Initialisation, prior to setting any of these bits to 1 (and thereby *enabling* interrupts of the specified kind), the software has to place information about what to do when an interrupt occurs into pre-defined locations in Main Storage. The information is in the form of a *new* PSW which is to be loaded into the CPU when the interrupt occurs. (Just prior to loading the *new* PSW, the CPU stores the *old* PSW into other pre-defined storage locations.)

There are five *classes* of interrupt, three of which can be disabled via bits in the current PSW (i.e. they are *maskable).*

The classes of interrupt are as follows:-

1)Maskable
- a)External interrupts (e.g. Interval Timer).
- b)Machine Check interrupts.
- c)I/O interrupts (from Channels)

2)Non-maskable
- a)Supervisor Call interrupts (generated by the SVC instruction).
- b)Program Check interrupts (generated when a problem program encounters a condition that makes it impossible for the CPU to continue the normal processing of the program).

Each class of interrupt is assigned its own *new* PSW location in Main Storage, and its own *old* PSW location in Main Storage. The locations are shown in Figure 5 on Page 14. Normally, the *new* PSW value will be *disabled* for maskable interrupts, so as to avoid "interrupt loops" where a subsequent interrupt occurs within a class before the interrupt-handler code has saved all the information associated with the previous interrupt. The (disabled) routines which are pointed to by the new PSW's, and which save the interrupted program's status (PSW, registers, etc), are usually called First Level Interrupt Handlers (or FLIH's).

Address	Purpose
24	External old PSW
32	Supervisor call old PSW
40	Program check old PSW
48	Machine check old PSW
56	Input/output old PSW
88	External new PSW
96	Supervisor call new PSW
104	Program check new PSW
112	Machine check new PSW
120	Input/output new PSW
The above addresses are all in decimal, all the locations referred to are doublewords.	

Figure 5: System /360 Interrupt PSW Locations in Main Storage
Note that System /370 introduced another, very important, interrupt class - the Restart Interrupt. The Restart old PSW is saved in locations 8-15, and the new PSW is taken from locations 0-7. This interrupt class is not maskable.

Once the FLIH has saved all the necessary information with respect to the interrupted process, it then usually transfers control to a Second Level Interrupt Handler (or SLIH) which actually deals with the event being signaled by the interrupt. When the SLIH completes, it normally transfers control to a *Dispatcher,* which then allocates the CPU to the highest-priority task in the system that is waiting for CPU (a *ready* task). This may, or may not, be the task that was interrupted, depending on what new information came in with the interrupt.

SLIH's can be either enabled or disabled, depending on their design. (Applications always run enabled - thereby ensuring that the Supervisor can pre-empt their access to the CPU when more urgent tasks become ready to run.)

7.1Example of an I/O Interrupt

The following series of figures takes us, step-by-step, through the "life" of an interrupt from an I/O device ...

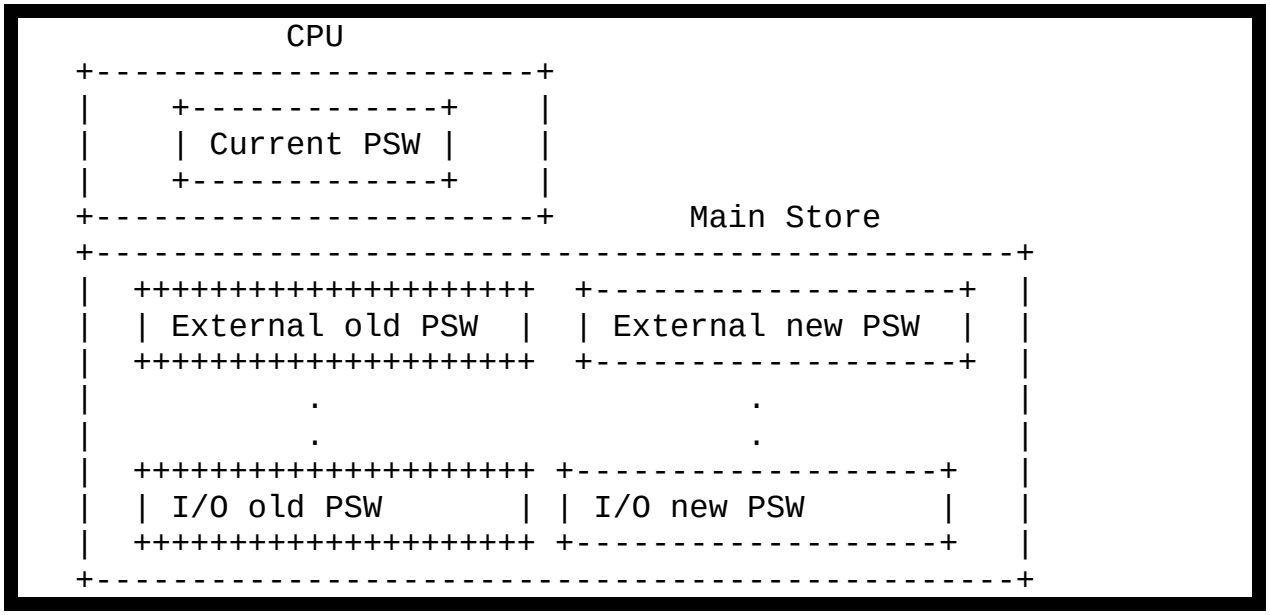


Figure 6: I/O Interrupt Example - Step 1
The system is executing an application, under control of the Current PSW. The operating system software has set up sensible values in the "new PSW" locations in low storage.

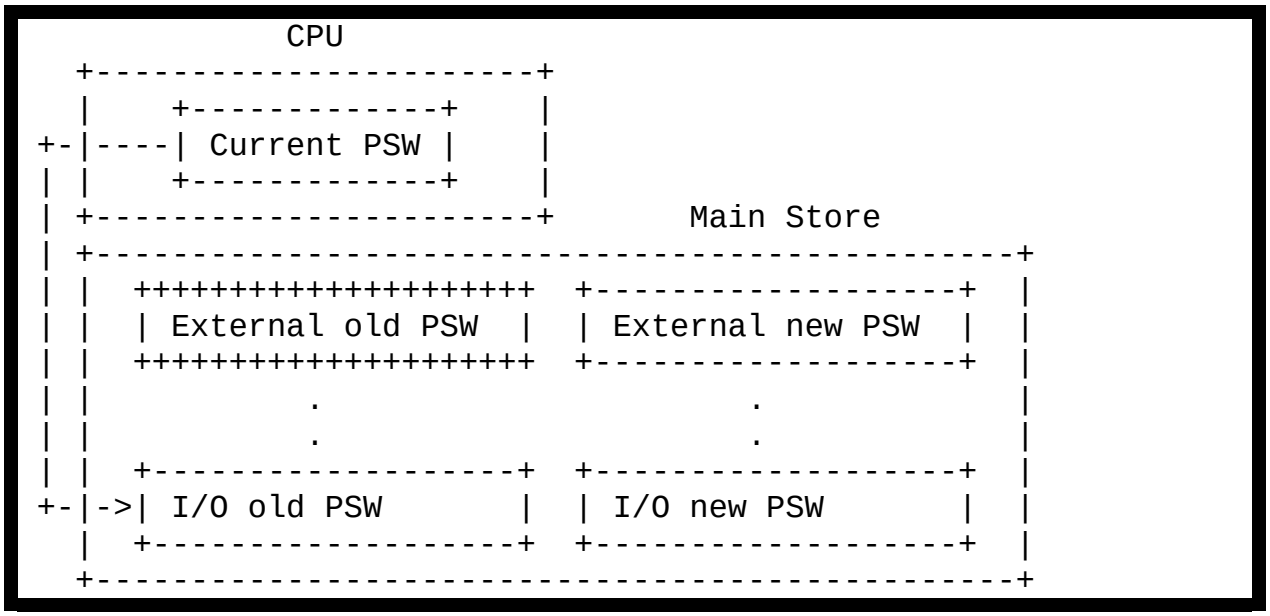


Figure 7: I/O Interrupt Example: Step 2(a)
An interrupt is received from one of the channels. The CPU stores the Current PSW into the I/O Old PSW.

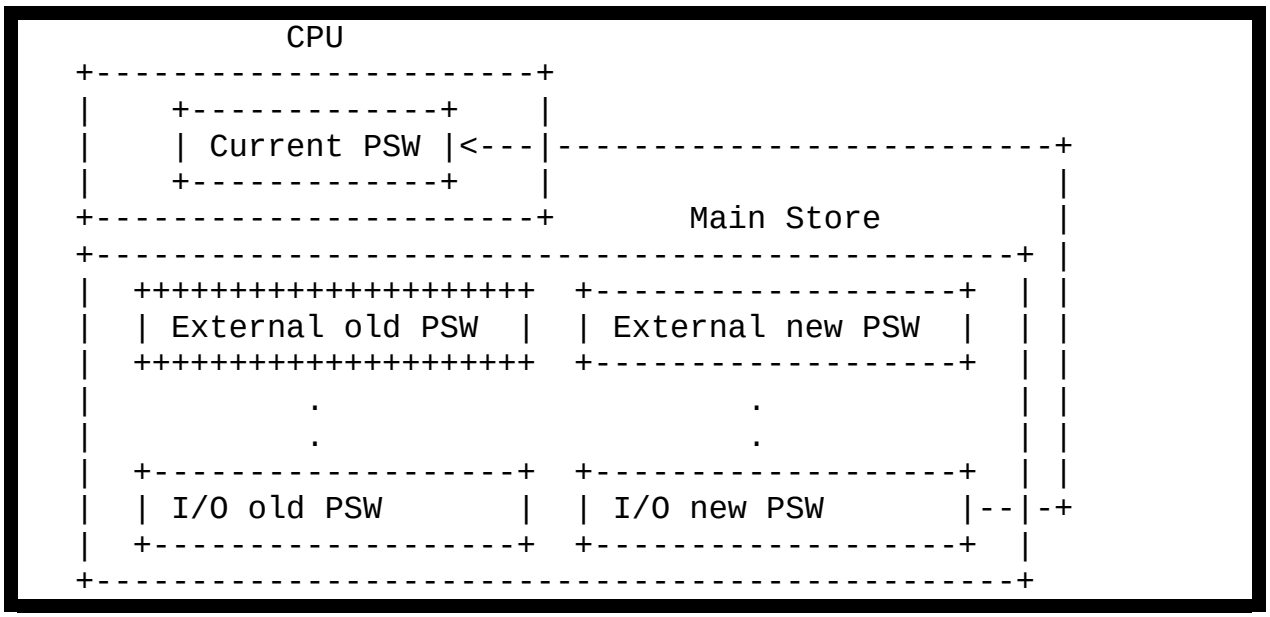


Figure 8: I/O Interrupt Example - Step 2(b)
The CPU then reloads the Current PSW from the I/O New PSW set up by the Supervisor in Main Storage.

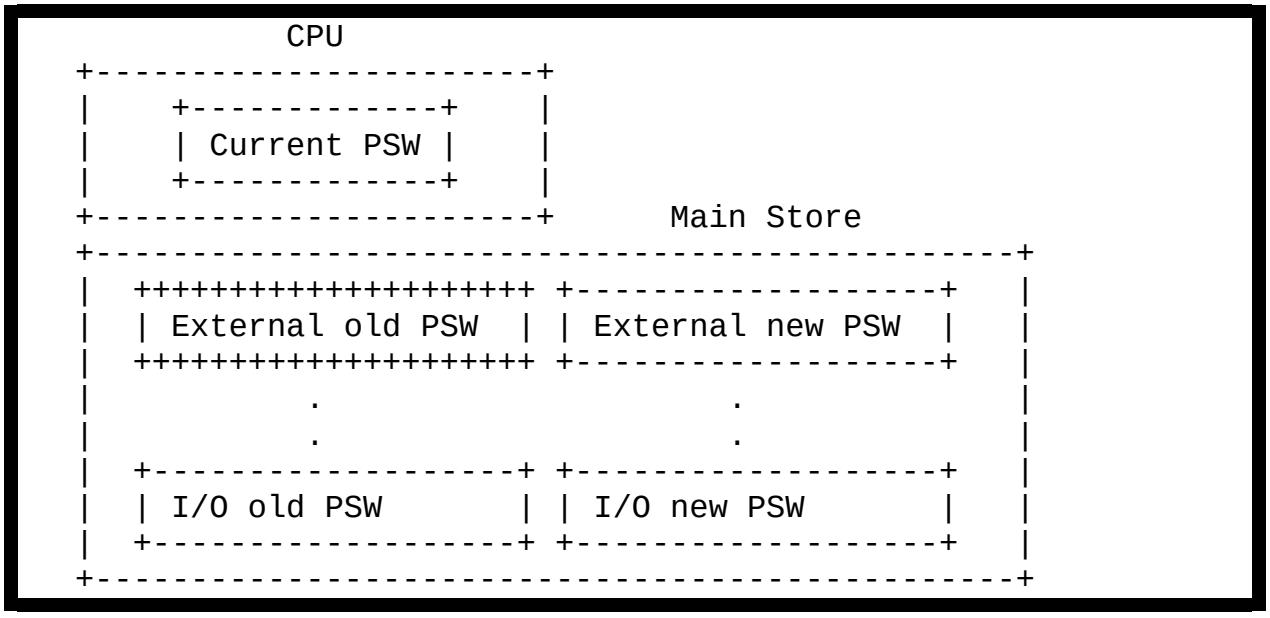


Figure 9: I/O Interrupt Example - Step 3
The CPU then goes into its usual fetch-and-execute cycle, thereby giving control to the I/O FLIH pointed to by the I/O New PSW.

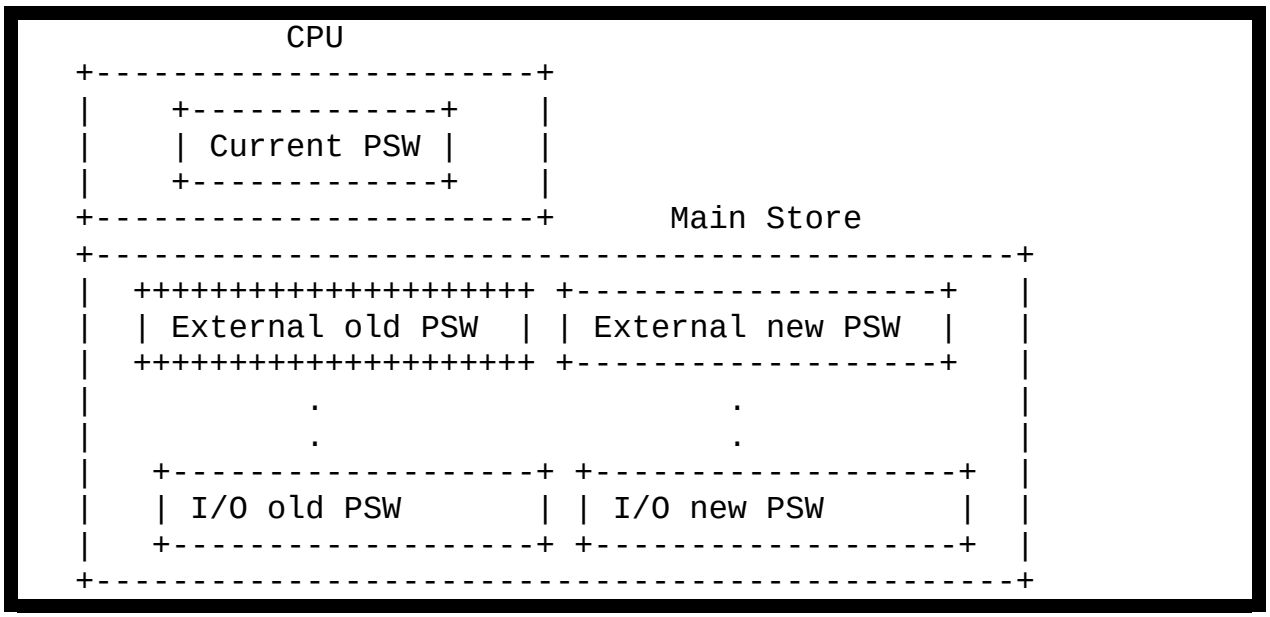


Figure 10: I/O Interrupt Example - Step 4
The FLIH saves the Old PSW and other status with respect to the interrupted task, and the passes control to the Supervisor to process the I/O interrupt.

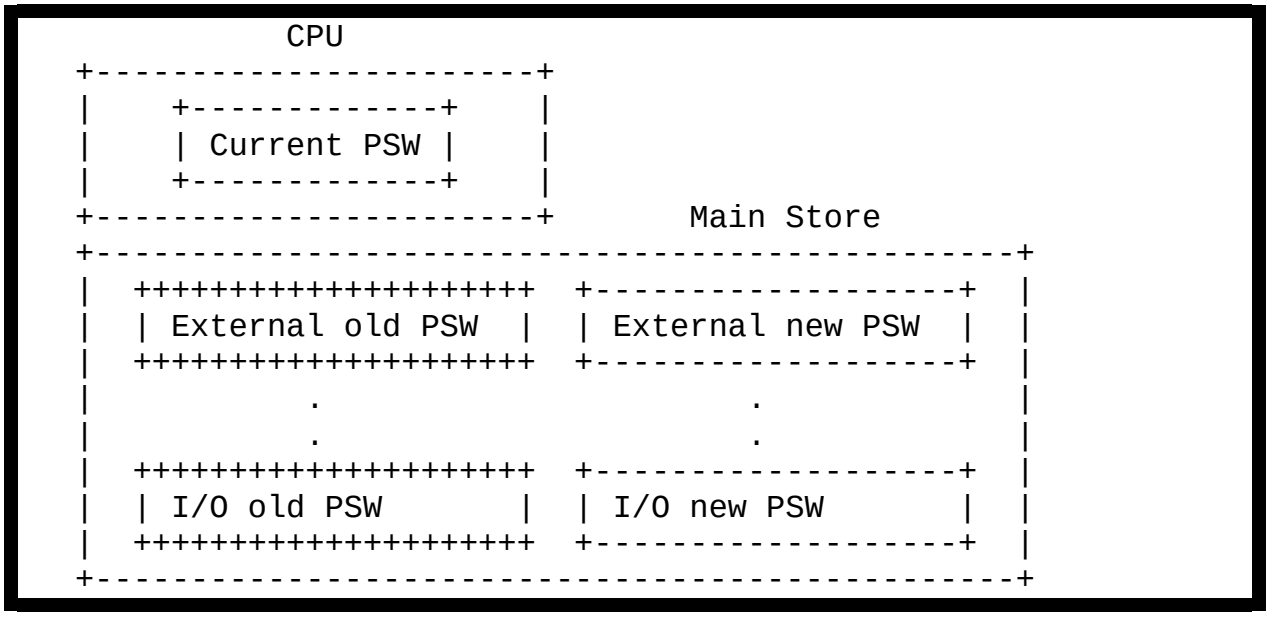


Figure 11: I/O Interrupt Example - Step 5
(Eventually) the Supervisor determines that there is no other more urgent task in the system waiting to run, and restores the status saved by the I/O FLIH, thereby resuming the interrupted task.

8.0Constraints to Growth (1)

/360 processors were very fast. Despite the development of multi-programming Supervisors such as OS/MFT and OS/MVT which provided advanced services like Access Methods (QSAM, BPAM, QISAM, etc) and Program Management (LINK, LOAD, XCTL, etc), along with spooling and Job Management systems (such as HASP and ASP), it was still very difficult to exploit the CPU to its full capability.

8.1The Storage Problem

The first, and most significant, problem was that the machines just didn't have enough Main Storage. With storage costing around $1.00 per byte, it was simply too expensive to purchase enough to cope with the level of multiprogramming needed to keep the processor properly utilized. Of course, it was well known that programs rarely made use of the storage they needed all at the same time - Program Management specifically provided services to allow applications to be loaded in "segments" as a program's logic proceeded from step to step. However, segmentation required careful programming, was prone to erroneous use, and was a programming overhead on the task "in hand". Furthermore, segmentation only addressed the efficient use of main memory by an application's *logic*. This same "locality of reference" applied equally often to an application's *data* areas - often significantly larger than the logic in terms of storage requirements.

A mechanism was needed for automatically allocating Main Storage to applications on an as-required basis, and automatically de-allocating the storage when that part of the application was idle. However, this had to be done in a way that was compatible with the majority of existing applications. There was already too much invested in existing /360 code to scrap it all and start again. The mechanism that was chosen became known as Virtual Storage.

8.2The Input/Output Problem

Seven channels weren't going to be enough. Given the potential multiprogramming level achievable with Virtual Storage (people were suggesting levels of ten, or even twelve, as being within reach), more I/O paths would be needed.

Also, DASD devices were becoming more prevalent, bringing with them a new problem for the channels. A DASD channel program to read or write a 4K block of data would typically require about 50ms to complete. Of this 50ms, only about one 5ms burst is actually required to transfer the data. About another 5ms is required to exchange control information between the channel and the control unit. The rest of the time (40ms) is spent waiting for the read/write heads to move to the correct cylinder (seek time), and waiting for the right data block to rotate round to the heads (latency). Clearly a maximum possible utilization of 20% on an expensive channel was not enough. A new kind of channel had to be designed, with the speed of a Selector, but with at least some of the multiprogramming capabilities of the Multiplexor. It became known as the Block Multiplexor channel.

8.3The Processor Problem

In order to manage Virtual Storage and the new channels, the CPU would have to keep track of more control information than could be held in the PSW. The solution to this problem was to add a new set of registers to the CPU, accessible only via Supervisor State instructions, called Control Registers. The architecture was extended to allow for up to sixteen Control Registers.

So that the new machines could be compatible with /360's at the *Supervisor* level as well as at the *application* level, two modes of control were required. The currently-active control mode is indicated by a previously unassigned bit in the PSW. The meaning of all the other bits in the PSW depends upon the setting of the mode bit.

The rather crude System /360 multiprocessor function was also enhanced into a properly manageable facility, allowing the development of commercial Multiprocessing Operating Systems such as MVS.

9.0The Architecture Matures.

Resolution of the Constraints to Growth required some very significant enhancements to the basic /360 architecture; sufficient to justify a new name. System /370 was the name chosen - probably intended to reassure customers that this was a compatible growth out of /360 and not something totally new. Indeed, by using a mode bit in the PSW, a remarkably high degree of compatibility was achieved.

As well as the vitally necessary changes to /360 that were required to allow customers to grow, many other extremely useful additions were made. Several of the more tedious application-level (Problem State) restrictions were lifted - such as the need to align the operands of various instructions onto integral boundaries; and new timing facilities were added to make overall system control and accounting easier for the Supervisor software.

However, we shall concentrate on the changes that made growth possible. From this presentation's point of view, these were the significant changes.

9.1Virtual Storage

Main Storage was expensive and under-utilized. A mechanism was needed that would provide the illusion that there was more application Main Storage available than was really the case. Typically, a 370/145 would require the illusion of about 2M of Main Storage in order to remain fully occupied when in reality only about 512K was available.

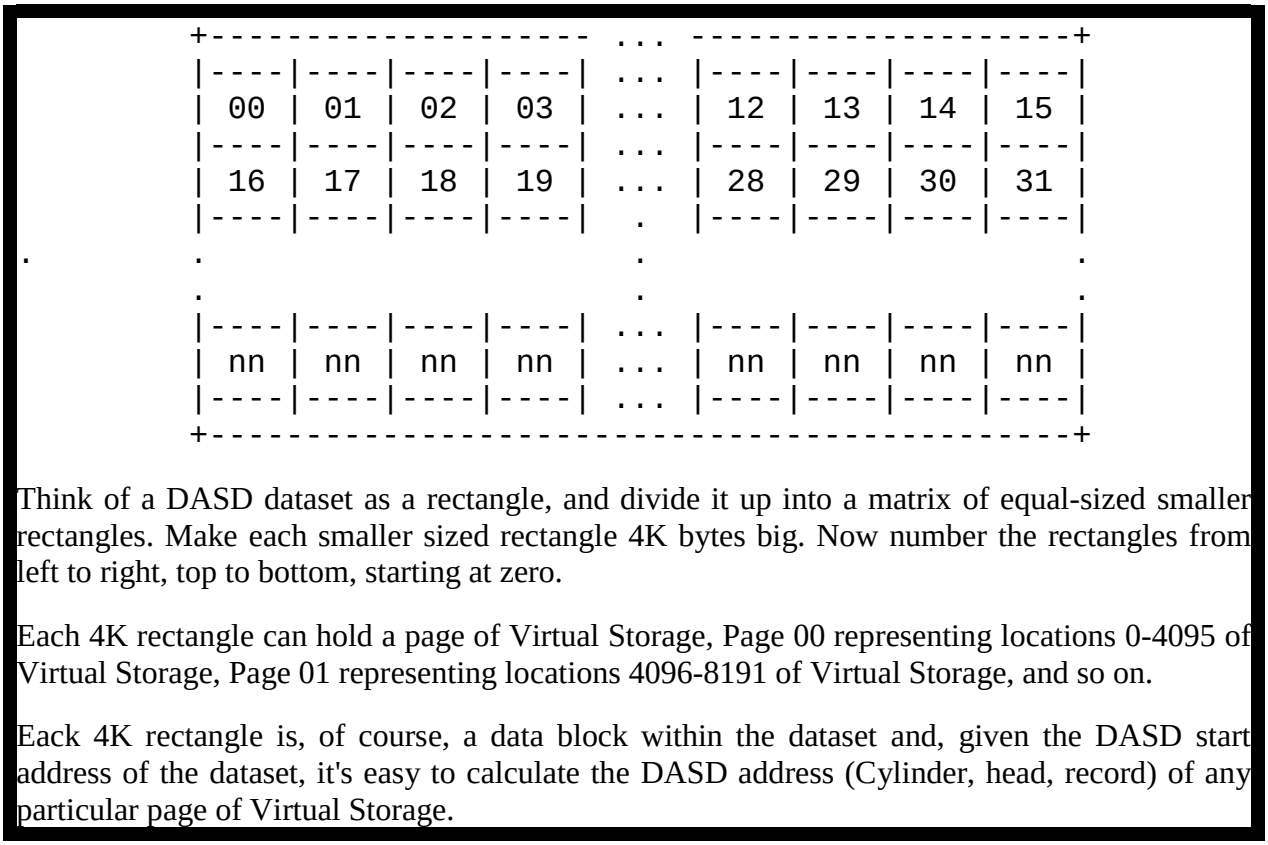
The illusion was achieved by imagining a Main Storage of the desired size, divided up into a number of fixed size blocks. (The /370 architecture allows for two block sizes - either 2K or 4K - under Supervisor control.) The earliest common Virtual Storage Supervisors - OS/VS1 and DOS/VS - used a 2K block. Later on, 4K became the standard and later releases of the 2K systems were changed to use 4K.

These "virtual storage blocks" required a name of their own and, because they were a bit like the pages in a book (where only two pages are visible to the reader at any one time, and an "interrupt" has to be serviced in order to access another page), the word "page" was chosen to describe them.

Having imagined a Virtual Storage, divided into pages, of sufficient size to meet our applications' needs, it's a small step to imagine Main Storage also divided up into pages of the same size, each one potentially capable of containing an "open" page of Virtual Storage. Of course, as there are far fewer Main Storage (real) pages than there are Virtual Storage (virtual) pages, we need somewhere to keep the "closed" Virtual Storage pages. We also need a control mechanism to keep track of which pages are "open" (in Main Storage), and which are "closed" (not in Main Storage).

Clearly, DASD storage is of sufficient capacity to be able to hold a complete Virtual Storage. (Why, a 2311 could hold a 2M Virtual Storage and still have 6M of space left for data - and a 2314 could hold 16M, the theoretical maximum Virtual Storage size!) DASD also provides ready access to each individual page of Virtual Storage as and when the page needs to be "opened" and "closed". See Figure 12.

As the Supervisor is going to be responsible for managing Virtual Storage, a large part of the Supervisor logic needs to be permanently resident in Main (or *Real*) Storage. The rest of Real Storage can be used to hold currently open pages of Virtual Storage.



Think of a DASD dataset as a rectangle, and divide it up into a matrix of equal-sized smaller rectangles. Make each smaller sized rectangle 4K bytes big. Now number the rectangles from left to right, top to bottom, starting at zero.

Each 4K rectangle can hold a page of Virtual Storage, Page 00 representing locations 0-4095 of Virtual Storage, Page 01 representing locations 4096-8191 of Virtual Storage, and so on.

Each 4K rectangle is, of course, a data block within the dataset and, given the DASD start address of the dataset, it's easy to calculate the DASD address (Cylinder, head, record) of any particular page of Virtual Storage.

Figure 12: Virtual Storage Mapped onto a DASD Dataset
This is a very simple "conceptual" mapping. Operating Systems use much more complicated structures, but they all embody the essentials shown in this figure.

Translation of virtual addresses is a task performed by the CPU, using the Dynamic Address Translation (DAT) logic. The Supervisor communicates with DAT using Control Registers, and tables which it builds and maintains in Real Storage (the Segment and Page Tables). Obviously, at IPL time these tables will not have been built, so the Supervisor must be able to initialize itself without DAT being active, and then signal the CPU in some way once it is ready to handle its share of the work involved in supporting Virtual Storage. The signalling mechanism is a bit in the PSW, known as the DAT (or *translate*) bit. If the DAT bit is on, the CPU translates storage addresses passed to it before processing the operands. If it's off, this translation is not performed.

Bits 8-12 of Control Register 0 specify to DAT the size of the page to be used (2K or 4K), and also the size of another structure used in virtual storage management - the *segment*. Segments are an intermediate structure which lies (in size) somewhere between individual pages and the complete Virtual Storage. They are used to simplify the work the Supervisor has to do in order to manage virtual storage. /370 architecture allows segments to be either 64K or 1M in size. A 64K segment is almost universally used by /370 supervisors. Looking at Figure 12, we could consider each *row* of the matrix as being a segment, the first (Segment 0) containing pages 00-15, the second (Segment 1) containing pages 16-31, and so on.

Bits 8-25 of Control Register 1 (also known as the Segment Table Origin register - STOR), together with six bits of zero appended to the right, point to the Segment Table built and maintained in Real Storage by the Supervisor. This table contains one entry for each segment of Virtual Storage defined by this Supervisor. A segment table for use with 64K segments has a maximum of 256 entries; one for use with 1M segments has a maximum of 16 entries. The segment table always starts at an address divisible by 64, and each entry in the table is 4 bytes long. The first entry describes Virtual Segment 0, the second describes Segment 1, and so on. Every segment, from zero to the highest virtual address in the Virtual Storage being described, *must* have an entry in the segment table. Virtual addresses above the highest address defined in "our" Virtual Storage don't need segment table entries because the length of the segment table (and therefore by implication the highest valid virtual address for this particular Virtual Storage) is defined to the CPU in bits 0-7 of CR1. Any attempt to reference a segment beyond the end of the segment table causes a program-check interrupt, just as if the segment were marked invalid (see below).

Segments can be either *valid* or *invalid*. An invalid segment has none of its pages available in Real Storage. The Supervisor tells DAT which segments are invalid by setting bit 31 (the segment-invalid bit) of the corresponding Segment Table entry to one. If, on the other hand, a segment of Virtual Storage is valid (bit 31 = 0), then bits 8-28 of the Segment Table entry, together with three bits of zero appended to the right, point to a *Page Table*, also maintained in Real Storage by the Supervisor. The Page Table starts at an address divisible by 8, and each entry in the table is 2 bytes long. The first entry describes Virtual Page 0 within the segment, the second describes Page 1, and so on.

Pages, like segments, can be either *valid* or *invalid*. An invalid page is not currently available in Real Storage. The Supervisor tells DAT which pages are invalid by setting the page-invalid bit of the corresponding Page Table entry to one. (This is bit 12 for 4K pages, and bit 13 for 2K pages.) If, on the other hand, a page of virtual storage is valid (page-invalid bit = 0), then bits 0-11 (4K page) or bits 0-12 (2K page) of the Page Table entry contain the page number of the Real Storage page frame currently assigned by the Supervisor to the specified Virtual Storage page frame. That is, the Virtual Storage is "open" at that page, which can be "read" by the CPU without any further action on the part of the Supervisor.

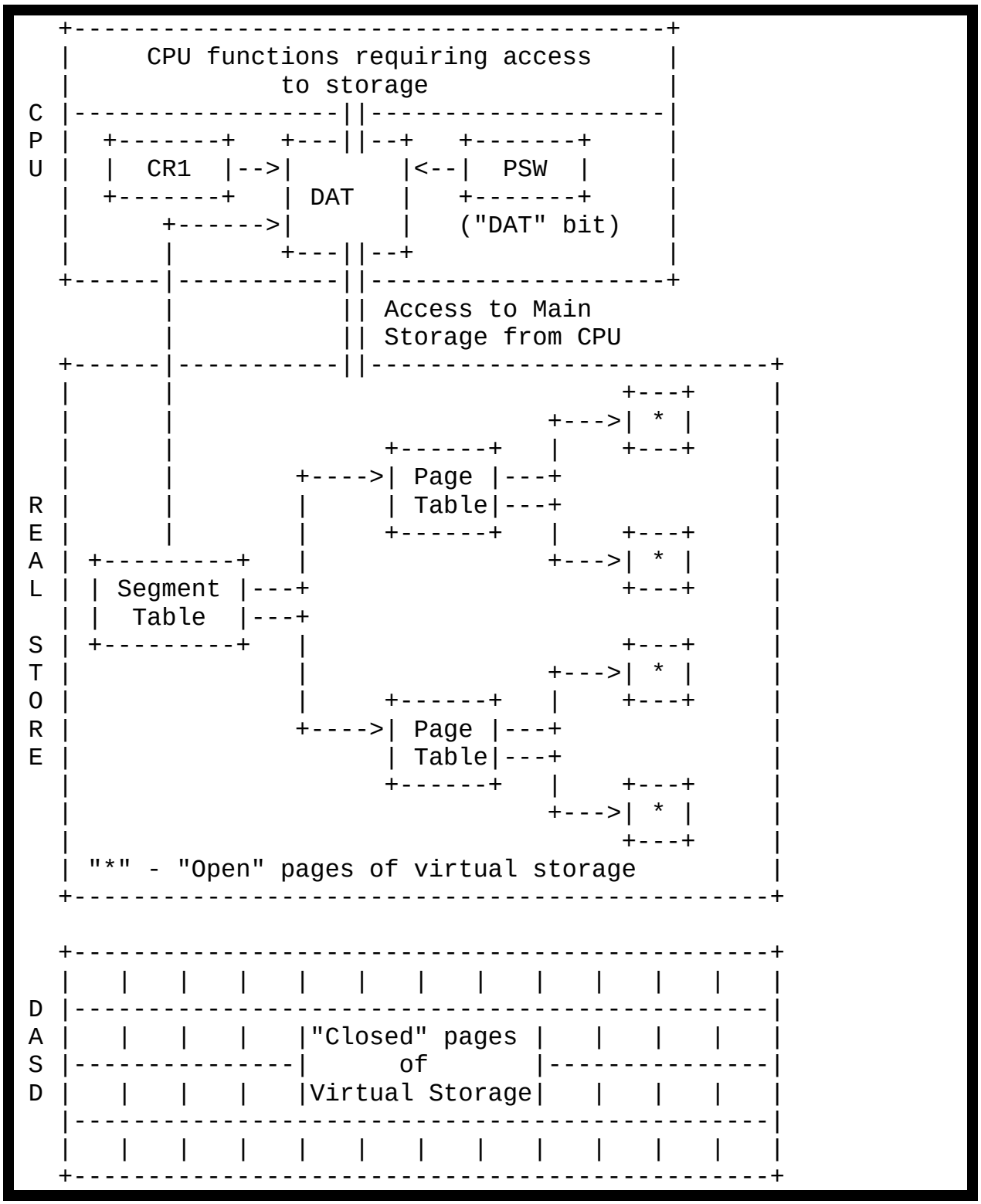
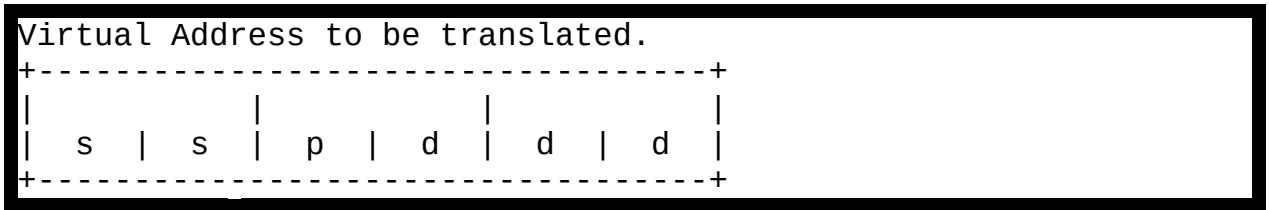


Figure 13: Dynamic Address Translation and Virtual Storage



ss	Segment number. Provides offset into Segment Table.
p	Page number. Provides offset into Page Table.
ddd	Byte displacement within page.

If segment and page table entries are both marked as valid, replace <ssp> with bits 0-11 of the Page Table entry to generate the corresponding Real Address.

Figure 14: Components of a Virtual Address (4K page, 64K segment)

When DAT is on, the CPU treats *every* address passed to it from non-privileged instructions as a *virtual* address, and attempts to translate it into a corresponding real address using the tables pointed to via CR1[4]. If it is unable to complete the translation (because either the segment-invalid or the page-invalid bit is on) then it signals a Program Check interrupt (called a segment fault or a page fault, respectively) to the Supervisor. Hopefully, the Supervisor has set up a Program Check New PSW that has the DAT bit set off and which points to a FLIH resident in Real Storage. Once the FLIH has saved the status information it passes control to the Virtual Storage Manager (VSM) who must assign a page of Real Storage from its pool of currently available page frames, read in the data in the corresponding DASD page frame, and then modify the Page Table entry accordingly. (In the meantime, of course, the Dispatcher has been assigning the CPU to other tasks whose pages are (hopefully) resident in Real Storage.)

The Virtual Storage Manager must also regularly "close" virtual page frames that have become idle, marking the Page Table entries as invalid and copying the data back onto the appropriate DASD page frame. It does this in order to maintain a pool of available Real page frames for use when dealing with page faults. To help the VSM in this task, DAT maintains two bits of information with respect to each page of Real Storage. These bits (two "spare" bits in the Storage Key) are the *reference* bit and the *change* bit. They are examined and turned off by the VSM using Supervisor State instructions, and turned on by a CPU or channel whenever it references or changes data held within the page in question.

The first Virtual Storage Operating Systems were "virtualised" versions of the /360 Operating Systems. The entire Supervisor was contained in the bottom pages of Virtual Storage which were permanently assigned to their corresponding Real Storage pages, thereby creating a "Virtual=Real" (V=R) area. Although this presence of the Supervisor within the Virtual Address Space was highly desirable for compatibility with earlier Supervisors (and is still present even in the latest MVS systems) it is architecturally completely unnecessary. Indeed, by changing CR1 to point to one of several Segment Tables, one could activate different Virtual Address Spaces in

[4] For privileged instructions (that is those instructions that are only allowed to execute when the CPU is in Supervisor State) sometimes the addresses are virtual, and sometimes they are real, depending on the instruction.

turn - each one up to 16M in size and completely independent of all the others - with the Supervisor totally inaccessible to any of them. This is, of course, the basic strategy of the CP component of VM.

9.2Changes to Channel Architecture

Once it had been decided to provide sixteen Control Registers to extend the control information previously held completely within the PSW, supporting more than seven channels was simply a matter of deciding which Control Register bits were to be assigned as Channel Mask bits to enable or disable interrupts from specific channels. Control Register 2 was allocated, with each bit (starting from the left), masking its corresponding channel. That is, bit 0 of CR2 masks Channel 0, bit 1 masks Channel 1, and so on.

Unlike System /360 POP[5], which specifically restricts the channel numbers that are valid in a device address (see Figure 4 on Page 12), System /370 POP specifically *does not* place any restriction on the channel number. While on one hand this implies that 256 channels could be controlled by a single CPU, the number of bits in CR2 effectively limits us to a maximum of 32.

It is unfortunate that the early designers of operating systems either didn't read /360 POP carefully enough or assumed that the high-order digit of the channel number would always be zero. The result of this assumption was to develop a widely held belief that device addresses were only three hexadecimal digits in size. This belief was embodied in all the systems of the time and led to a great deal of re-coding when the hardware arm of IBM was able to start shipping processors with more than sixteen channels. Indeed today's MVS/XA still retains a three-digit "device number" limitation because the impact of changing to four digits would be too great. VM/SP "bit the bullet" with HPO Release 3.6 and all current VM/HPO and VM/XA systems use four digits to address a device.

The second I/O problem - dealing with DASD I/O which transferred data in (relatively) short bursts during (relatively) long-running channel programs - was resolved by developing a new type of channel - the Block Multiplexor Channel. Like the Multiplexor channel, the Block Multiplexor contains several subchannels, each of which is capable of supporting a concurrently executing channel program. The difference comes when a control unit requests access to Main Storage.

On a Multiplexor (sometimes now referred to as a Byte Multiplexor in order to differentiate it from the Block Multiplexor), access to storage is granted, and a few bytes of data (usually just one or two, some printers manage as many as six) are transferred within a "time-window" required by the device. If access is not granted within this time-window, a "Channel Overrun" condition occurs. This will require special recovery action to be taken, such as the execution of an Error Recovery Program (ERP) in the Supervisor, or operator action.

On a Block Multiplexor, when a control unit is ready to access storage, it tests to see if the data path to the channel is busy. If the path is available, the control unit signals the channel that it wants to use the data path, and a conversation takes place. If the path is busy, the control unit

[5] "POP" stands for "Principles of Operation" - the manual that describes the system architecture, and from which practically all the information in this presentation is drawn. There is a separate POP manual for each major level of the architecture. See "" for full details.

waits until the next time the device is ready to transfer data, and then tries again. (In DASD terms, "readiness" corresponds to the required block of data moving under the read/write heads; if access to the channel is not possible at the correct moment the control unit has to wait for a full revolution of the DASD platter before it can try again. This condition is often referred to as Rotational Position Sensing (RPS) Miss. RPS Miss can be a significant contributor to DASD I/O delay on Block Multiplexor channels that are over 30% busy.) The crucial point is that the recovery is automatic.

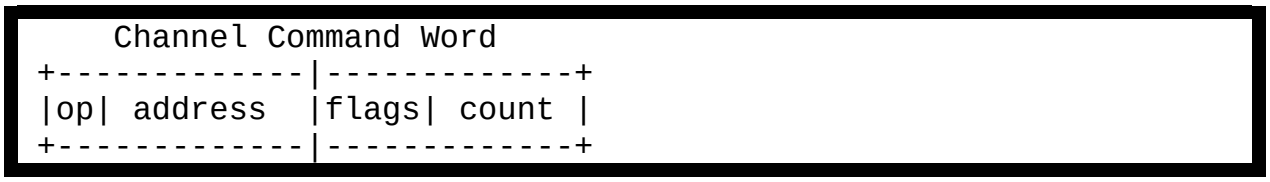
The Block Multiplexing mode of operation is known as *burst* mode and Byte Multiplexors are also capable of operating in burst mode. However, burst mode on a Byte Multiplexor is the exception rather than the rule.

The last significant change in channel architecture was brought about by the demands of supporting I/O to applications that were running in Virtual Storage. When an application asks for a block of data to be transferred into or out of what it sees as Main Storage (but which is really Virtual Storage) it will expect the data to be transferred to or from contiguous storage locations. The Supervisor overhead that would be involved in arranging that several contiguous pages of Virtual Storage occupied several contiguous pages of Real Storage while a data transfer took place would be substantial. (Remember, DAT is a *CPU* function - channels don't need or want to know about Virtual Storage.)

So that channels could read or write a single block of data from non-contiguous pages of Real Storage, channel logic was extended to allow Channel Indirect Data Addressing. A flag bit in the CCW indicates to the channel that the CCW address points not to the data area (as was always the case with /360 channels), but instead to the first Indirect Data Address Word (IDAW) in an Indirect Data Address List (IDAL). The number of entries in the IDAL depends on the size of the block being read or written. There is one entry for each 2K "block" of storage that is referenced (see Figure 15on Page 27).

This now makes the Supervisor's job much easier. When an I/O operation is requested by an application the Supervisor "merely" has to ensure that the relevant contiguous pages of Virtual Storage are resident somewhere in Real Storage (and that they remain locked there until after the channel program has completed!), construct an IDAL using the Real Storage addresses, point the channel program at the IDAL, and then issue the Start I/O instruction!

Practically all /360 and /370 Operating Systems allow applications to define requests for data via channel programs which are built by the application without any consideration for the possibility that they might be running in Virtual Storage. When the Supervisor receives such a channel program for execution it must analyze the channel program, CCW by CCW, and construct an equivalent *real* channel program for execution by the channel. This process, commonly known as *CCW translation* can be a major contributor to the Supervisor Overhead in a Virtual Storage system. The only time CCW translation can be bypassed is when the application is run in an area of Virtual Storage that the Supervisor permanently maintains in Real Storage at the equivalent Real Storage addresses (that is, a V=R area).



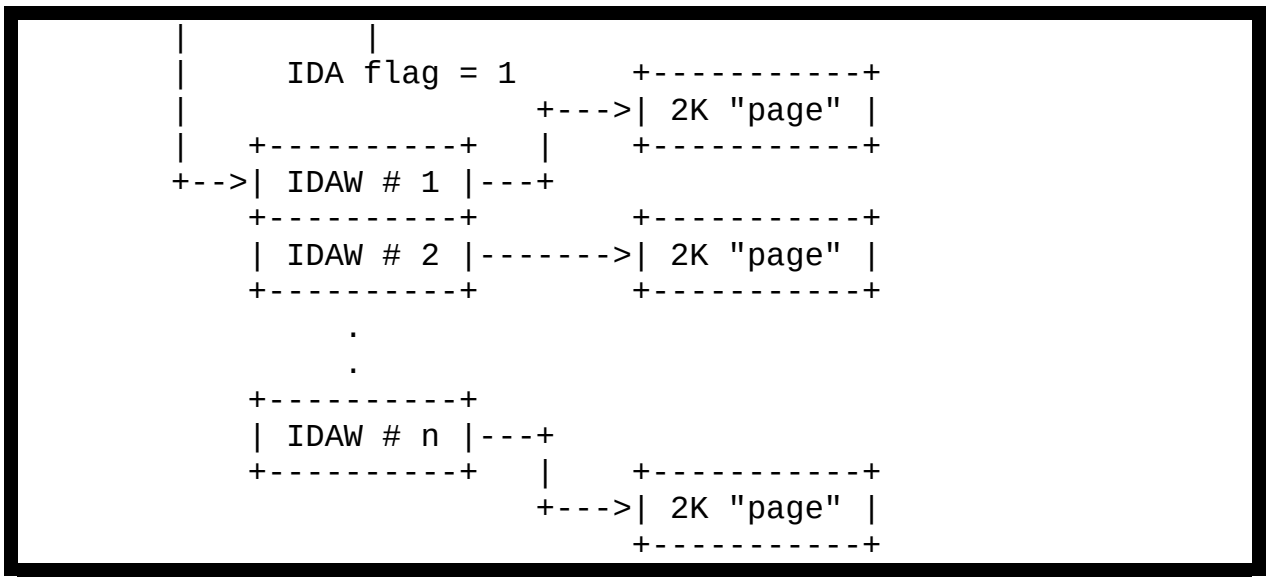


Figure 15: Channel Indirect Data Addressing
By using a 2K "page", IDALs are compatible with either 2K or 4K Virtual Storage page formats. (Interestingly, even though 370-XA allows only 4K pages, this vestigial 2K page support remains, and IDAWs continue to refer to storage in units of 2K, even in the latest ESA /370 systems.)

9.3CPU Changes

Extended Control (EC) mode, together with the sixteen Control Registers, provided ample additional control capability to support the new channels and Virtual Storage. Indeed, even in the latest System/370 Principles of Operation, only thirteen Control Registers are defined.

In the table that defines Control Register assignment (in Chapter 4 of POP) there is a curiously coy footnote to the effect that bits 13, 30 and 31 of Control Register 0, and bits 0-30 of Control Register 6, "are assigned to functions not described in this publication". There is no hint or suggestion as to where the assigned functions *are* described - the innocent reader is left alone with his speculations. In actual fact, they are used by VM/370 to communicate with the VMA hardware feature (CR6) and by Virtual Machines running under VM to communicate control information to CP services - PAGEX, IUCV and VMCF (CR0). Perhaps the hardware specialists who control the contents of Principles of Operation are ashamed to admit that they've actually assigned some aspects of their beloved architecture to that Maverick VM Operating System!

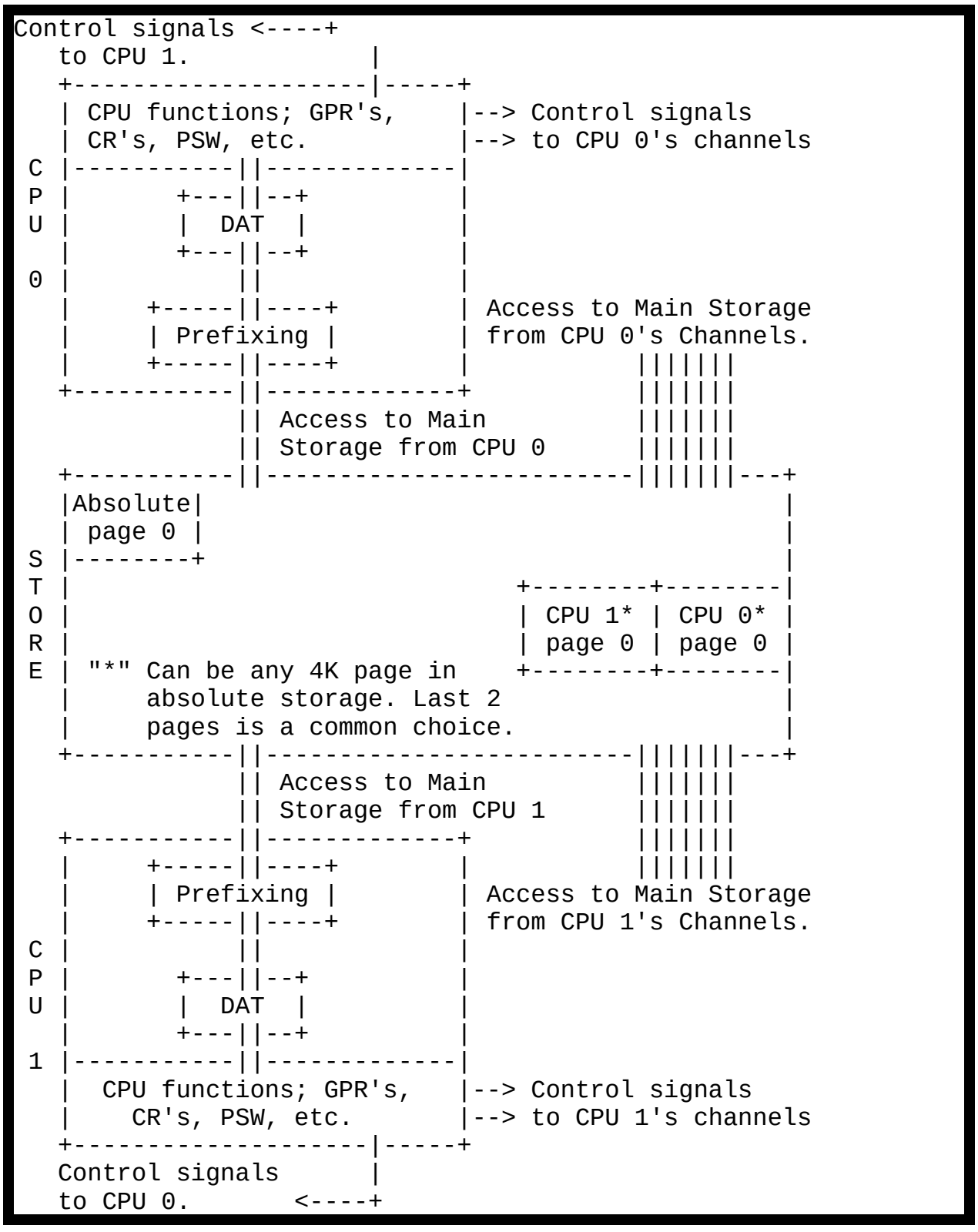


Figure 16: Basic Central Components of a /370 (Multiprocessor)

As well as EC mode, the architecture was also extended in anticipation of new demands for CPU

resource now that the I/O and storage bottlenecks had been opened up. It seemed likely now that the general-purpose demand could grow to the point where a single CPU could not keep up; a second processor would be required. (Also, a configuration with two processors could keep going on just one processor while the other was offline for maintenance or repair. Was this when IBM created their now famous RAS acronym, standing for, "Reliability, Availability and Serviceability"?)

All the basic elements needed for multiprocessor operation were already in existence in System /360, but System /370 refined these into a more manageable set of facilities, and the first strategic commercial MP Operating Systems to run on IBM mainframes were System /370 systems.

Along with a second processor comes a number of complications, such as the need for *two* low-store communication areas and (potentially) another full set of up to 32 channels. Most of the control problems associated with multiple processors are resolved by a technique known as *Prefixing*. Real Storage addresses are no longer Real, they are re-defined as the storage addresses as seen by a particular CPU. The addresses permanently assigned to Main Storage are now known as *absolute* addresses. Mostly, real and absolute storage addresses point to the same locations in Main Storage. The exception is the 4K block of storage pointed to by absolute address zero, and the 4K block of storage whose absolute address has been loaded into the CPU's *Prefix Register* using the Set Prefix (SPX) command. The real addresses of these two 4K blocks are their *opposite* absolute addresses. This allows each CPU in a complex to have its *own* "page zero" containing its interrupt vectors, CAW, CSW, interval timer, and so forth. Theoretically, one processor in the complex could set its prefix to zero, and use absolute page zero as its real page zero. In general, this is not done, and absolute page zero is used to keep information that is common to all CPU's. (There is a specific implementation of MVS-and-VM together on one real machine, known as PMA, in which MVS *can* use absolute page zero as its real page zero - but beyond knowing that the combination exists, I have no further knowledge.)

Each CPU in an MP configuration has a unique 16-bit *CPU address*. This seems to suggest that, architecturally, it would be possible to configure as many as 65,536 processors into a single complex although, as there's only 4096 4K pages in a 16M Main Store, we may have trouble finding somewhere to keep all the "page zeros" in such a configuration! Perhaps it's fortunate that IBM has never shipped the capability for more than 2 CPUs in a System /370 complex.

Along with SPX, several other instructions were provided to allow the CPU's in a complex to signal to each other and to synchronize their access to Main Storage whenever such synchronization was necessary.

Perhaps it's worth emphasising that each CPU has its *own* set of registers, its *own* DAT logic, and its *own* timers. Co-ordinating the activity of these largely independent engines is a substantial software task. One of the trickiest tasks to manage effectively is the driving of I/O requests. I/O devices which are attached to only one channel must be driven via the CPU connected to that channel. Where an I/O device is connected to two or more channels, attached to both CPU's, the Supervisor needs to ensure that the workload is reasonably well balanced so that the entire subsystem performs as well as possible. Also, remember, when a device is attached to more than one channel, the Supervisor may need to be aware that the *one* device has *several* addresses.

It should really come as little surprise to discover that even the best tuned MP configurations

rarely perform better than 1.8 times as well as the equivalent UP, and many systems programmers would be delighted to achieve an MP factor of 1.6!

A technique known as Channel Set Switching, which allows one CPU to "take over" another CPU's channels, was developed to allow continued access to devices connected only to one channel when that channel's CPU was offline (RAS, again), but it's a complex and expensive process in terms of the software support it requires.

The increase in complexity that would be involved in adding a third or fourth CPU suggests that one would be fortunate to obtain an MP factor of 2.2 with 3 processors, and 2.8 with four processors. The architecture could just about handle 2, but some radical changes were going to be needed before we could usefully have machines with three or more engines.

10.0Constraints to Growth (2).

10.1The Storage Problem.

Once applications were liberated from the tyranny of having to fit into Real Storage, their demands for Virtual Storage zoomed. This, of course, led to a "knock on" demand for Real Storage by the operating systems in order to support the applications' demands.

By killing off 2K paging (which was by now out of favour anyway), two spare bits in the Page Table entries were pressed into service as storage address bits, allowing Real Storage to grow up to 64M. At the same time, the granularity of storage protection was changed from the 2K boundary to the 4K boundary. This reduced overhead by removing the need to manage two separate keys for each individual page of real storage.

Dual address-space (DAS) facility allowed specially coded applications to make use of two virtual storages of up to 16M each - subject to various restrictions - but something more radical was soon going to be needed.

10.2The Input/Output Problems.

CCW's only had a 24-bit address field. Although addresses over 16M could be accessed via IDAW's, something was going to have to be done.

Tying channels to particular CPU's, and requiring the I/O Supervisor to keep track of each of the individual components in a data path (Channel, control unit, device) was consuming CPU cycles that would be better used executing application code. The I/O subsystem needed to be physically and logically separated from the CPU's - only then would 3-way, 4-way, and even 6-way MP configurations make sense.

10.3The Processor Problem.

The architectural changes needed to resolve the I/O problems were going to make it impossible for the new processors to be compatible with /370's at the *Supervisor* level. It was even probable that a significant percentage of applications would require modification; particularly those that had either deliberately bypassed the formal *Application Interfaces* (API's) defined by the Operating Systems, or were coded before the API's had been defined.

Transition from the old to the new was via a hardware (microcode) emulator built into the new machines. The machines could be initialised in "crippled mode" - when they emulated System /370 Architecture - or in "native mode" - when they delivered the new architecture. However, *programmed* mode switching such as had happened on the change from /360 to /370 wasn't even attempted.

11.0The Architecture Undergoes Radical Surgery.

The changes between System /370 and System /370 Extended Architecture are far more substantial, and far less compatible, than the changes from /360 to /370. On the whole, changes from /360 were either completely new features or compatible extensions to existing features. This was not to be the case with the changes from /370 to /370-XA. The entire I/O architecture was to be replaced with a radically new, utterly different, scheme of things. This was going to have a major impact on a large proportion of the Supervisor. Furthermore, architectural changes were going to be visible at the application program level. It is a credit to IBM's technicians that (at least for MVS users) the changeover has been so smooth for most customers. It is possibly of even greater credit to IBM's sales force that such radical and incompatible changes could be presented to the customer as mere architectural extensions!

Once again, as well as the changes of immediate interest in the context of this presentation, there were a number of changes designed to help the Supervisor manage the system. Included in these changes is the TRACE instruction, which acknowledges the significant effort most production Supervisors put into maintaining diagnostic information.

Another incidental change was the removal of the Interval Timer at locations 80-83 in Main Storage. The feature that had given VM/370 so much trouble in the early days of running production Guest Operating Systems (such as OS/VS1) under VM was not going to cause problems for 370-XA!

11.1Extended Storage Addressing.

"Everyone" knows that 370-XA means 31-bit addressing. I personally find the degree of exclusive concentration on this really rather minor change slightly puzzling. The stage had obviously been set for 31 or 32-bit addressing with the advent of System /370, when bits 32-39 of the EC mode PSW were defined as "unassigned"[6].

Extended addressing potentially impacts every /370 program already in existence. Too much code already exists that, for instance, depends upon the Load Address instruction to set bits 0-7 of a register to zero, for a change of this kind to be universally, arbitrarily and unconditionally imposed. A "mode bit" was needed, settable by the application program, that defined which addressing mode the processor was to use. Bit 32 of the PSW was chosen, and a repertoire of new instructions was introduced to set, test, save and restore the addressing mode. Principles of Operation was re-written to define, for each affected instruction, the differences in behaviour of the instruction when executed in 24-bit and 31-bit mode. For all problem-state instructions, the behaviour of the instruction in 24-bit mode matched the behaviour of the equivalent System /370 instruction, thereby allowing compatibility at the application program level.

---

[6] In fact, Amdahl 470 and 580 processors with more than 16MB of storage have 31-bit real addressing as standard, even when running in /370 mode. The Amdahl VM/Software Assist package makes full use of this capability to optimize its execution on these machines. Undoubtedly there were other 31-bit /370-type processors of which I am unaware.

11.2Changes to the I/O Subsystem

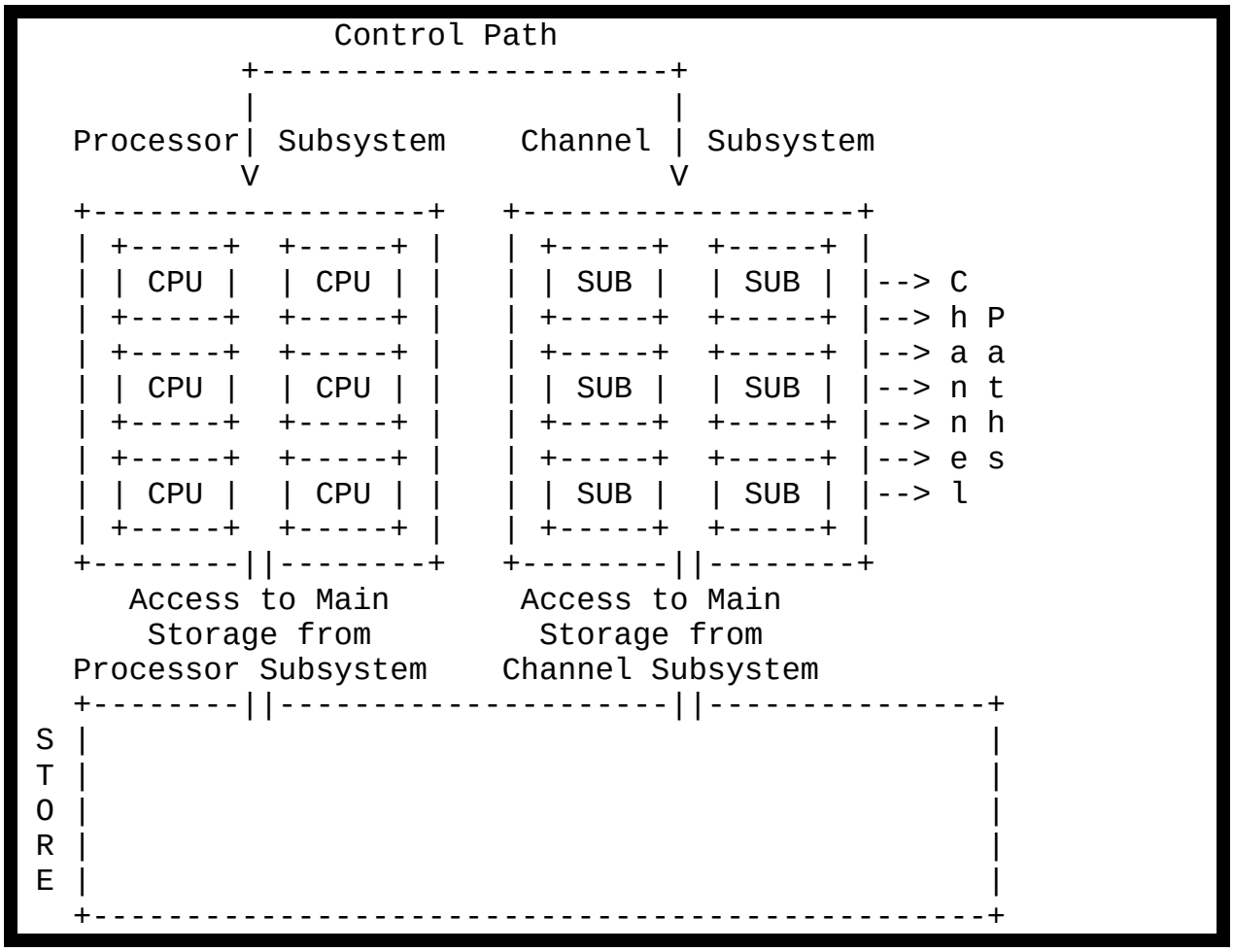


Figure 17: Separation of CPU's and Channels in /370-XA

Channels were logically separated from Central Processors. Any CPU can be used to receive interrupts from any device in the system for I/O operations started by any CPU.

Architecturally this is achieved by dividing up the machine's components into *subsystems*, and arranging for communication to take place at the *subsystem* level rather than, as previously, at the *component* level.

370-XA POP talks about the "Channel Subsystem"; it does not explicitly state that all the CPU's are gathered together into a "Processor Subsystem" (as shown in Figure 17), but this seems to me to be a totally satisfactory way of picturing the architecture.

Inside the Channel Subsystem, some major changes took place. Channels, as such, ceased to exist. Each device in the configuration was assigned its own *dedicated* subchannel with its own, unique, subchannel number - a 16-bit number, allowing a maximum of 65,536 devices per Channel Subsystem. Devices are addressed by the Processor Complex using the subchannel number assigned at configuration time. (Don't confuse subchannel number with *device number*, also unique and also assigned at configuration time. Device numbers are chosen by human beings and are quite arbitrary; subchannel numbers are assigned by the configuration program

and always run contiguously from 0000 to the largest number required.)

During configuration, each subchannel is advised of up to 8 *channel paths* it may use to access the device that it manages. These channel paths connect the Channel Subsystem to the Control Units of the devices in exactly the same way a Channels connect to Control Units in System /370. There can be up to 256 channel paths in a Channel Subsystem.

When the Processor Subsystem asks for a channel program to be executed, the Channel Subsystem notifies the appropriate subchannel, which then selects *any* channel path from the set of available channel paths to communicate with the Control Unit / Device. It is quite possible for *different* paths to be used at different stages in the execution of the channel program. For example, a DASD Control Unit operating in Block Multiplexing mode can be advised that it's allowed to reconnect onto any one of several channel paths when it's ready to transmit or receive data. This feature (Dynamic Path Reconnect) allows 370-XA systems to achieve much higher channel utilization without serious performance degradation from RPS miss.

The key difference in 370-XA from the I/O Supervisor's point of view is that the tactical details of path selection and management for a particular I/O operation have been completely delegated to the Channel Subsystem. The Supervisor has one point of contact with each device instead of the (possibly) four that he had with /370.

The Operating System's I/O Supervisor still remains in charge of the Channel Subsystem - there is no loss of authority. For instance, it can limit a subchannel's choice of channel paths using commands like Modify Subchannel. It continues to define the I/O strategy dynamically but now simply communicates the strategy to the Channel Subsystem for the subsystem to implement rather than having to manage all the details itself. This, of course, then frees up CPU cycles for more commercially rewarding activities.

11.3CPU Changes

There's certainly nothing "central" about the CPU's in a 370-XA system - they're merely components within a processor complex! (See Figure 17 on Page 33.)

Architecturally the changes to a 370-XA CPU, when compared with a /370 CPU in a /370MP configuration are relatively quite minor. CPU management remains completely the responsibility of the Supervisor software, and the management mechanisms (prefixing, CPU signalling, and so forth) are pretty much unchanged.

Of course, the new channel architecture made several detail changes, for instance CR2 is now unassigned and eight bits of CR6 are assigned to I/O interrupt subclass masks. (The I/O Supervisor dynamically assigns interrupt subclasses to subchannels using the Modify Subchannel command.) VMA no longer needs CR6 because something far better is now available - the *architected* Virtual Environment!

In terms of the function it delivers, nobody could describe Interpretive Execution as "minor", but it falls so far outside the original mainstream of 370-XA that POP fails to mention it at all - even in passing. It also falls beyond the scope of this presentation, but I have made a note of the Interpretive Execution manual's reference number in "", so you can find out more if you wish.

With 370-XA, the size of segments and pages became fixed. Segments are always 1M, pages are

always 4K. Curiously, the five bits in CR0 that defined the translation format in System /370 are still assigned as "Translation Format" bits - but only one combination of bit values is defined as valid!

12.0Conclusion.

The next step in the evolution has already been taken. Enterprise Systems Architecture /370, with its ability to allow one application to address a "sheaf" of address-spaces, each up to 2GB in size, is with us today. I, though, have failed to keep up and, apart from the most general of comments, am not at all qualified to speak about ESA/370.

I would comment that ESA/370 is a *superset* of 370-XA. That is, the change from 370-XA to ESA/370 is, for most of us, going to be far less painful than the change from System /370 to System /370-XA. In fact, it should be about the easiest architecture step to take that IBM has come up with yet!

(It is interesting that, as with the change from /360 to /370, the ESA/370 features can be turned on or off by the Supervisor. The processor is initialised in ESA/370 mode if one wants to run either 370-XA or ESA/370 systems, and an ESA/370-capable Supervisor can then dynamically turn off the new features while running older, incompatible applications.)

As well as not having any worthwhile comment to make on ESA/370, I see that I have also been delinquent in failing to make any mention of Expanded Storage or Vector Processors - both significant contributors to performance in their own ways. I am sure that there are many other features of all the architectures that I have failed to touch on. For these omissions I ask your forgiveness. I hope that, despite these shortcomings, you find this presentation to have been useful.

13.0References.

13.1Primary References

The primary references for all the information contained in this presentation are the various IBM Principles of Operation manuals that have been published over the years. At the time of writing, all these manuals are available from IBM. I particularly recommend GA22-6821 for beginners to read - life was so much simpler then.

Don't be put off by the title of GA22-6974 - it's a fascinating document that takes all the "black magic" out of how Channels and Control Units exchange information - well worth half a day of your time.

Order no.	Title
GA22-6821	IBM System/360 Principles of Operation.
GA22-7000	IBM System/370 Principles of Operation.
SA22-7085	IBM System/370 Extended Architecture Principles of Operation.
SA22-7200	IBM Enterprise Systems Architecture/370 Principles of Operation.
SA22-7095	IBM System/370 Extended Architecture Interpretive Execution.
GA22-6974	IBM System/360 and IBM System/370 I/O Interface Channel to Control Unit Original Equipment Manufacturers' Information.

13.2Secondary References

Order no. **Title**

My copy of the following manual is dated September 1967, and I have absolutely no idea whether or not it's still available. However, I've found it an extremely useful secondary reference and maybe you can get hold of a copy somewhere. Try some of the "old men" of your site - the retired Systems Programmers who are now in senior management.

C20-1646 A Programmer's Introduction to the IBM System/360 Architecture, Instructions, and Assembler Language.

I have not read the following manual, but I include it as a likely source of information for anyone who is interested in delving deeper into the mysteries of Vector Processing.

SA22-7125 IBM System/370 Vector Operations.

Other sources of information include back copies of IBM System Journals. Unfortunately, with the one exception shown below, I cannot cite specific articles owing to an ill-advised housekeeping exercise carried out a couple of years ago; but I recommend a scan through the contents pages of Journals from the mid-Sixties to the mid-Seventies, if they are available to you.

Journal IBM Systems Journal Volume Three, Numbers Two & Three | 1964