Final Design Specification For the MCS65E4 Microprocessor

Written by: Yang / Goodman / Mathys
Revision: 1.1
Date Released: 10-MAY-82

This document is submitted with the understandings that it contains information which is confidential in nature and is not to be revealed to anyone without written permission from MOS Technology, Inc.

Revision History

Rev #	Date	Description of Revision
0.0	2-Oct-81	Original release
1.0	1-MAY-82	General clean-up and reorganization. Rewrite of software architecture description

Table of Contents

1.0 Introduction

This specification contains a detailed description of all aspects of the MCS65E4 microprocessor development project, beginning in Section 1 with a review of the project goals and a discussion of the market toward which this chip is directed. It is hoped that these discussions will lead to greater understanding of the goals of the project on the part of everyone involved.

Section 2 contains a description of the MCS65E4 interface. This is followed by a description of the internal architecture of the MCS65E4 (Section 3), including the register organization, the internal buses and the organization of the control store. Section 4 contains a detailed description of the MCS65E4 software architecture (addressing modes, instruction set, etc.).

1.1 Review of MCS65E4 Project Goals

Before entering into a detailed discussion of the MCS65E4, it would be useful to briefly review the major factors which have influenced the design of this processor system. Understanding these factors will be particularly important for anyone involved in the design verification stage of this project.

Although the MCS65E4 is equipped with a 'compatible' mode in which it is capable of executing software which was written for the MCS6502, the MCS65E4 is not designed to be upward compatible with the 6502 family of 8-bit microprocessors. The primary reason for this is that the basic design considerations behind the 6502 processor differ greatly from those described below for the MCS65E4 processor. This is true in spite of the fact that the 6502 has reached a dominant position in the microcomputer market, one of the target markets for the MCS65E4.

To put the 6502 architecture into perspective, it should be noted that when this design effort began, microprocessors were viewed primarily as replacements for random logic in the design of controllers. The 6502 was optimized toward this application. To this end, significant emphasis was placed on minimum system configurations and on minimizing device and system cost. This was accomplished through the use of such things as page zero addressing, 8-bit index registers, multiple-function support devices, and generally simplified system interfacing.

Many of the characteristics of the 6502 which were designed to maximize its performance as a random logic replacement would seem to limit its performance in high-end microcomputer systems. In spite of this fact, low cost and ease of use has allowed the 6502 to become a dominant factor in this market. These are the features which will be retained in the MCS65E4. At the same time, the architecture of this 'next generation' processor will be designed to assure maximum performance in microcomputer systems at the lowest possible cost.

Modern high-end microcomputer systems exhibit several features which can greatly influence the design of a processor optimized

for this application. In particular, all such systems are controlled by a sophisticated operating system. In many cases, components of this operating system are swapped into and out of memory as required. Most such systems support several user's programs in a 'multi-task' environment, reallocating the available memory from system to user or from one user to another as required.

There are several important problems inherent in this type of system. The first is memory protection. It is important that the operating system be protected from the user's programs and that the user's programs be protected from each other. In addition, the software should be 'relocatable' since the physical address space in which the program will be located is generally determined at execution time.

In addition to the above, it is assumed that most microcomputer programming will involve the use of a high-level language. Therefore, the software architecture of the processor must be designed to minimize the time required for both compilation and execution of such languages.

Finally, it should be noted that even the most powerful processor is wasted if it is absorbed in I/O handling a large portion of the time. For this reason, the system level problems of interrupt, DMA, etc. must be handled in a manner which maximizes the amount of time which the processor has available for 'computing'.

All of these factors have had a strong influence on the design of the MCS65E4. However, the design described below addresses each of these factors in a manner which provides maximum performance within well-defined chip size constraints. The architecture described in this document can be built into a device which will be well within 'state of the art', providing a combination of device cost and performance which should allow it to assume the dominant position in the micro- computer market now held by the 6502.

1.2 Summary of MCS65E4 capabilities

The following is a brief listing of the principal features of the MCS65E4 family of microprocessors.

1. 8, 16 or 32-bit Data Bus.

2. 24-bit Address Bus.

3. ALU processes 32 bits of data for each processor cycle.

4. No internal data registers visible to the programmer. All operations are 'memory-to-memory'.

5. Internal operand registers allow processing of multi-byte operands.

6. 'Generic' Op Codes, i. e., the Op Codes do not specify

the format of the data fields.

7. 'Self-defining' data structures, i. e., most data is accessed through tags and descriptors. However, the ability to directly access and manipulate byte, double byte and triple byte fields is provided to facilitate the generation of descriptor and pointer addresses, etc.

8. On-chip hardware and microcode support for many operating system functions.

9. Hardware support for:

   a. Error Detection and correction.
   b. Virtual memory.
   c. Prioritized and vectored interrupts.
   d. Floating point data types.
   e. Decimal (BCD) data types.

1.3 Terminology

1.3.1 Introduction

The architecture of the MCS65E4 contains a number of very important concepts which are unique to the world of microprocessors. To assure the accurate transfer of information, therefore, this section introduces what is hopefully a clear, consistent terminology which will be employed throughout this document.

1.3.2 Process

The 'process' is one of the key concepts in the MCS65E4 architecture. In general, a process can be described as a self-contained combination of software and data. The address limits within which a process must execute are defined by information stored in an internal Process Base Register for the lower limit and in a Process Limit Register for the upper limit. Special hardware within the MCS65E4 assures that a process does not access any memory locations outside of the address space defined by these two registers.

There are several important process characteristics which affect the execution of software within the MCS65E4. The most important is that all processes are totally relocatable, i. e., an MCS65E4 program will execute in exactly the same manner no matter where it is located in the 16 mega-byte address space. In addition, an active process can be suspended, and can be moved within the address space of its caller without affecting subsequent execution.

There are three types of processes within the MCS65E4 architecture. These are the Kernel, the Operating System and the User process. Each exhibits characteristics which reflect its position in a well-defined hierarchy. The term 'Kernel process' refers to the lowest level in the set of processes which forms a

complete MCS65E4 system. The processor enters this mode through the chip reset function or through system calls and traps which occur in the higher level processes. Within the Kernel, the processor can call either a higher level operating system process or a User process. These higher levels of operating system can continue to call additional processes until a User process is encountered. This hierarchy of processes is described in detail below.

Within this specification, the terms 'Kernel process' will be used to refer to process level 1 in which both the Kernel flag and the User/Supervisor flag are set. The term 'Operating System Process will refer to those higher level processes in which the User/Supervisor flag is set but the Kernel flag is cleared. This can be summarized as follows:

Process	Kernel Flag	User/Supervisor Flag
Kernel	1	1
Operating System	0	1
User	0	0

1.3.3 Op Code

The term 'Op Code' refers to the first byte of each instruction. This byte specifies the operation to be performed (Add, Subtract, etc.) and the format of the instruction. However, it does not specify the type of the data (Real, BCD, etc.) which is to be manipulated by the instruction.

1.3.4 Operand

The term 'operand' refers to that portion of the instruction which contains the information necessary to access a single data field. The first byte of the operand specifies the manner in which the desired data field is to be accessed. Specifically, the data can be located in an internal register, it can be in the instruction (immediate data), or it can be accessed through the normal data referencing mechanism described below.

1.3.5 Instruction

The term 'instruction' refers to the combination of Op Code and Operands which are accessed under direct control of the Process Program Counter to cause a complete execution sequence to take place within the processor.

1.3.6 Descriptor

Within the MCS65E4 architecture, the 'data descriptor' acts as the primary means by which the processor determines the format and location of a data field. The term descriptor refers to all of the information required to access a data field. The components which

make up a descriptor are:

1. Descriptor Header.
2. Address Reference Information.
3. Auxiliary Information.

The operation of the descriptor is described in detail in Section 4.

1.3.7 Ordinal

The term 'ordinal' will be used to refer to the three-byte unsigned binary fields which are used to store logical addresses, offset addresses, etc. within the MCS65E4 architecture.

1.3.8 Static Data, Dynamic Data

During the discussions of process organization and execution within the MCS65E4, the terms static data and dynamic data will be used to differentiate between process variables which retain the same format for the life of the process and those which are created and abolished while the process is being executed. The most important characteristic of these two types of data is that the amount of memory required by the static data will not change during execution of the process. Dynamic data, however, consists of variables which cannot be assigned fixed amounts of memory during compilation of the process software because the memory requirements for these variables will only be known at run time.

1.3.9 Physical Address

The term 'physical address' will be used to specify a position in the 16-megabyte address space which the MCS65E4 can access. These are the addresses which appear on the pins of the processor.

Throughout this document, the physical address is assumed to be the 'default.' Therefore, if an address type (physical, logical , etc.) is not specified, it can be assumed to be a physical address.

1.3.10 Logical Address

One of the most important aspects of the stand-alone nature of a process is that all addressing within the process software is self-contained and is completely independent of the physical memory locations in which the process resides. All addresses generated during execution of the process software are assumed to be offsets from the address contained in the Process Base register. For example, if a process whose base address is 044B00 (HEX) were to specify an address of 0177 (HEX), the physical address which would be accessed is 044C77 which is obtained by adding 0177 to 044B00.

This characteristic of addressing within the MCS65E4 brings up the concept of the logical address. In this document, the term logical address will be used to refer to the position of a memory location within the address space of a process. In the above example,

therefore, the logical address would be 0177. It should be noted that all software execution within the MCS65E4 is performed within the context of a process. For this reason, all memory locations have both a physical and a logical address. The physical address remains fixed by the system logic. However, the logical address of each memory location is entirely a function of its position within a process. This will be illustrated in the example below (See Figure 1.1).

To assure accuracy, this document will utilize the phrase 'within process (process name)' whenever a logical address is specified. Also, a memory location which is outside of the limits of a process is assumed to have no logical address within that process, i. e., the logical address is assumed not to exist.

1.3.11 Page Address

There are many aspects of the MCS65E4 architecture which assume an eight bit organization. For example:

1. Op codes are eight bits wide.
2. The minimum addressable data field is eight bits wide.
3. Offset addresses can be zero, eight, sixteen or 24 bits.
4. Both the base and limit for a process are specified in 256-byte increments.

As a result, it will be useful to utilize the term 'page address' to identify the location of a 256-byte page. Throughout this document, the Page Address will be specified by the upper 16 address bits with the low order eight bits identified by XX. For example, Page Address 01E4XX identifies the page whose upper sixteen address bits are 01E4. This page includes addresses 01E400 through 01E4FF.

In addition to the Page Address, the phrase 'address on page (page number or name)' will be used to specify an eight bit address within a page. For example, address 01E43A can be identified as address 3A on page 01E4XX.

The term 'Base Page' will be used to refer to the lowest order page within a process. This is the 256-byte block of memory whose page address is contained in the Process Base register. Similarly, the term 'Limit Page' will be used to refer to the 256-byte block of memory whose page address is contained in the Process Limit register. The range of addresses which are available to a process extends from address 00 on the Base Page through address FF on the Limit Page.

1.3.12 Offset Address, Relative Address

All data addressing within the MCS65E4 is accomplished by adding a displacement to a memory address. This can be divided into two specific forms of addressing. These are Offset Addressing and Relative Addressing. These two differ primarily in the manner in which the memory address and the displacement are specified.

Within the MCS65E4 architecture, the term 'Offset Addressing' will be used to identify an addressing operation in which the offset is specified in the instruction and the memory address is contained in a base register. Only positive offsets are permitted when accessing through Offset Addressing. The base register can be either one of the on-chip process registers (TOS, BAS, PRM, LMT) or any three-byte set of addresses in the Base Page.

To assure accuracy, this document will utilize the phrase 'offset from register (register name)' whenever an Offset Address is specified. In addition, whenever an external base register is established in the process Base Page, this base register will be identified as 'EXT (n)', where n is the page address of the start of the base register. For example, if addresses 15-17 on the process Base Page are to be treated as an external base register, this base will be identified as EXT15. Finally, it will be assumed that a memory location which cannot be accessed through a base register has no offset address relative to that register. This will be true, of course, for any memory location outside of the process. Even more important, it will also be true for all memory locations with a lower physical address than that contained in the register since negative offsets are not permitted while accessing data via base registers.

In addition to Offset Addressing, the MCS65E4 utilizes a similar addressing mode in which the memory address is not contained in a base register and in which both negative and positive displacements are permitted. This is termed 'Relative Addressing'. Within Relative Addressing operations the memory address can be either the contents of the program counter or the address of a data descriptor. This is described in detail in Section 4 of this specification.

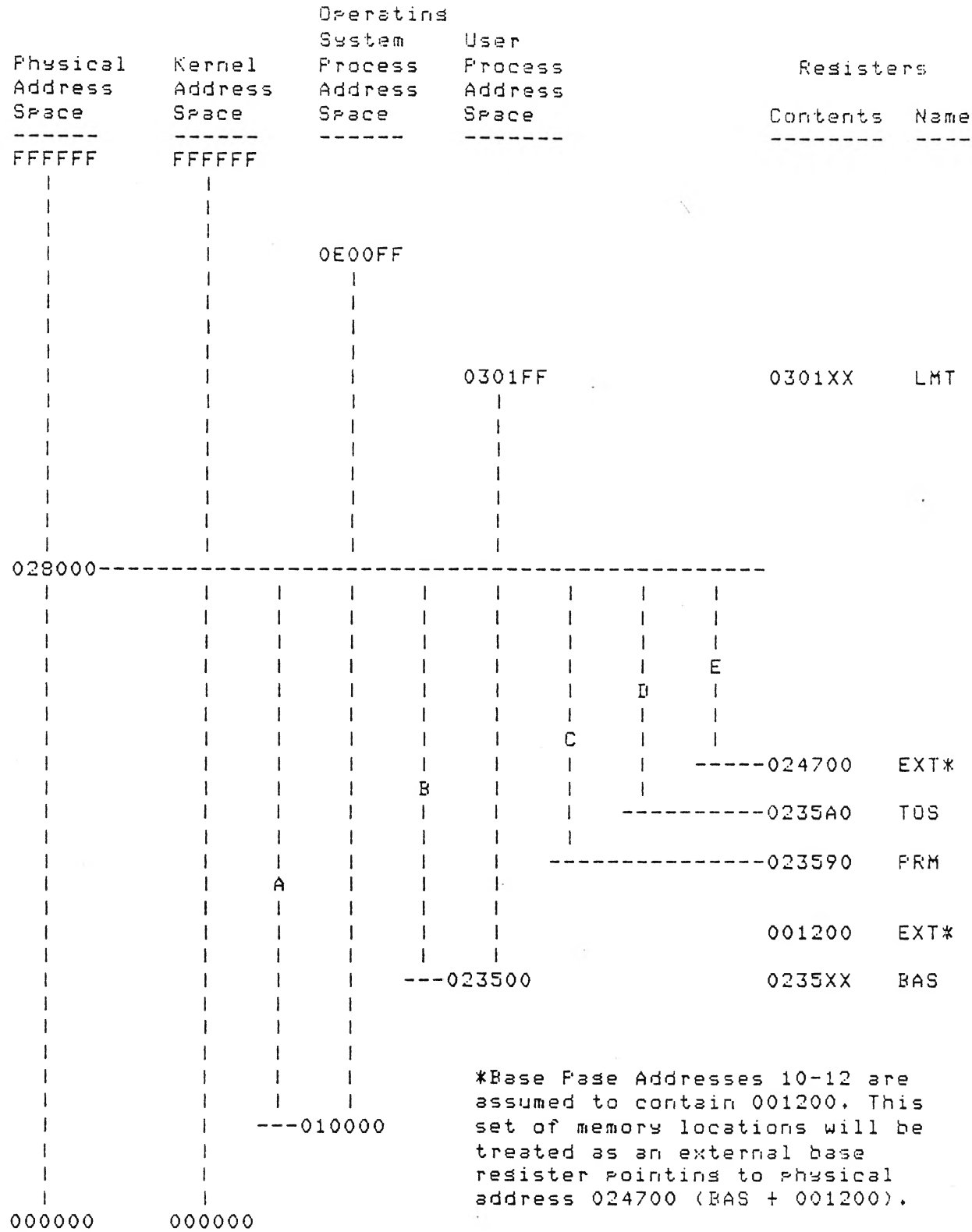


FIGURE 1.1- Initial Configuration for Addressing Example

1.4 Example of Addressing within the MCS65E4 System

The addressing concepts outlined above can be clarified by example. This will be accomplished by describing the addresses associated with a memory location which is contained within the address space of a User process. This User process is assumed to have been called by an Operating System process and is therefore at level three in the hierarchy. Any memory location located in this User process can be accessed by each of the lower level processes. Therefore, the memory location being discussed below will have a single physical address, but will have a logical address within the Kernel process, within the Operating System process and within the User process. In addition to these three logical addresses, the memory location will have a number of Offset Addresses during any period that the MCS65E4 is executing one of these three processes.

The diagram above illustrates the memory map of a multi-task system in which the three processes reside. The Kernel is assumed to cover the entire 16-megabyte space. The Operating System process (which was invoked by the Kernel) is limited to the range of addresses from 010000 to 0E00FF. At the same time, the User process is assumed to reside initially within addresses 023500 to 0301FF. The memory location which will be examined initially will be 028000, which is within the range of addresses allocated to the User process. Figure 1.1 illustrates this configuration.

As described previously, each memory location has a single physical address. For the memory location being examined initially, this physical address is 028000. In fact, since the base of the Kernel is always at address 000000, a memory location's logical address within the Kernel is the same as its physical address. Within the Operating System process (level two in the hierarchy) the logical address of this memory location is 018000, which is the displacement between the physical address of the base of the Operating System process and the physical address of the memory location itself. At the same time, this memory location has a logical address within the User process. This is obtained by subtracting the physical address of the memory location (028000) from the physical address of the process base (023500). The resulting logical address is 004B00.

To allow the offset addresses for this memory location to be specified, it is necessary to first specify the contents of the registers which can be used as a base for addressing the memory location. This will be illustrated by assuming the existence of two addressing registers, termed the Primary Base Register and the Top of Stack Register. At the same time, it will be useful to assume that addresses 10-12 within the Base Page contains 001200 and will be treated as an External Base Register. This provides three internal registers (including the Process Base register) and one external base register which can be used to access data.

To illustrate the offset address, assume that the MCS65E4 is executing the User process, the Primary Base register (PRM) contains 023590 and that the External Base Register (logical

addresses 10-12 within the user process) contains 001200. The Top of Stack Register (TOS) is initially at 0235A0. Under these conditions, the Offset Address of memory location 028000 relative to PRM is 004A70. Likewise, the Offset Address to TOS is 004A60 and the Offset Address to the External Base is 003900.

During the execution of the User process software, it is possible to modify the contents of the addressing registers introduced above. Doing so, of course, modifies the Offset Address of each memory location in the process relative to that register. It may in fact eliminate the Offset Address since only positive offsets are valid. This can be illustrated by assuming that PRM is set at logical address 000040 within the User process. Doing so sets the register contents to 023540. At this point, the Offset Address of memory location 028000 becomes 004AC0. However, if the PRM is set so that it points to address 029000 (logical address 0005B00 within the User process), this register can no longer be used to access address 028000 and therefore, this memory location no longer has an offset to the PRM register.

It should be noted that all memory locations outside the limits of a process have no logical addresses within that process. Likewise, memory locations outside of a process which is being executed have no Offset Address relative to the internal or external base registers since these locations cannot be accessed by these registers.

2.0 Description of the MCS65E4 Pin Functions

2.1 Introduction

The initial versions of the MCS65E4 will be available in a standard 40-pin dual-in-line package. This is made possible by multiplexing the address, data and bus status information onto a set of 24 pins. The pin configuration is as follows:

Function	# of pins
1. Address Bus Middle/ Address Bus Low (A9/A1-A16/A8)	8
2. Address Bus High/ Data Bus Lo (A16/DB0-A23/DB7)	8
3. Bus Status/ Data Bus High (IACK/DB8-MIC/DB15)	8
4. Row Address Strobe (RAS)	1
5. Column Address Strobe (CAS)	1
6. Chip power (VDD,VSS)	2
7. Oscillator	2
8. Bus Clock (BCLK)	1
10. Memory Ready (RDY)	1
11. Interrupt Input (INT)	1
12. Reset (RES)	1
13. Write Enables (WEL, WEH)	2
14. Bus Error (BERR)	1
15. Hold (HLD)	1
16. Instruction Intercept (II)	1
TOTAL	40

Each of these sets of pins is described in detail below.

2.2 Address Bus Middle/Address Bus Low (A9/A1-A16/A8)

The low order sixteen address bits (above A0) are multiplexed onto eight pins in a manner which is compatible with industry standard 64-Kbit dynamic RAMS. These lines enter the high impedance state for external DMA operations (see HOLD).

2.3 Address Bus/Data Bus/Bus Status (A16/DB0-A23/DB7, IACK/DB8-MIC/DB15)

The high order eight address bits and the bus status bits are multiplexed with the bi-directional data bits. During memory write operations, the timing for these signals is the same as for the low order sixteen address lines. For a memory read operation, the MCS65E4 output drivers enter the high impedance state and the memory devices place data onto these lines.

The high order address bits are normally stored in external latches to be used as chip selects for the memory and I/O devices. These signals are strobed by RAS as are the bus status bits. The bus status bits are used to control specific functions such as interrupt and DMA.

Bit	Status Function
S1	Interrupt Acknowledge (IACK)
S2	Hold Acknowledge (HOLDA)
S3	Last Instruction Cycle (LIC)
S4	I/O Reset (IORES)
S5	Processor Instruction Fetch (INST)
S6	Processor Data Fetch (DAT)
S7	Refresh Cycle (REF)
S8	External Microcode Fetch (MIC)

2.3.1 Interrupt Acknowledge (IACK)

The IACK bit goes high to signal the Interrupt Controller that it can place the active interrupt request information on the low order eight bits of the data bus. This operation is described in detail in Section 4.4.4.13.

2.3.2 Hold Acknowledge (HOLDA)

The Hold Acknowledge bit goes high to indicate that the processor will enter the HOLD state at the end of the present cycle. During the HOLD state, the RAS and CAS signals continue to run and the bus status signals are generated by the processor during RAS. However, no data, address or write enable (WEL, WEH) information is generated and the corresponding drivers remain in the high impedance state at the appropriate time.

2.3.3 Last Instruction Cycle (LIC)

The LIC bit goes high to indicate that the current cycle is the last cycle of an instruction execution sequence. This is used in conjunction with bus arbitration logic in multi-processor systems to control access to shared resources.

2.3.4 I/O Reset (IORES)

This bit goes low to cause the system I/O devices to be reset. This occurs when a System Reset instruction is executed. Causing the RES input signal to go low does not cause this bus status bit to go low. This allows resetting the processor without effecting the peripheral devices.

2.3.5 Processor Instruction Fetch

This bit goes high to indicate that the address on the address bus comes from the Processor Program Counter and that the data being fetched from memory will be placed into the input queue.

2.3.6 Processor Data Fetch

This bit goes high to indicate that the address on the address bus was generated as the result of an instruction execution.

2.3.7 Refresh Cycle

This bit goes high to indicate that the current cycle is a memory refresh cycle.

2.3.8 External Microcode Fetch

This bit goes high to indicate that the current cycle is an external microcode fetch cycle.

2.4 Row Address Strobe (RAS)

The Row Address Strobe is a clock signal used primarily to latch the middle eight bits of the address into external latches. These can be discrete TTL latches for interfacing to peripheral devices or to conventional static memories. In most cases, however, they will be located in the dynamic memory devices. In addition to the middle byte of the address bus, this signal indicates the presence of valid data on the high order address lines and on the Bus Status lines. RAS will be held low by RDY but will continue running during a HOLD operation.

2.5 Column Address Strobe (CAS)

The Column Address Strobe is primarily used to latch the column addresses (low order eight address bits) into external latches. This signal is also used to indicate that valid data is present on the data lines during a memory write operation and to enable the memory output drivers during a memory read operation. This signal is synchronous with the BCLK signal. The CAS signal is held low by RDY but will continue running during a HOLD operation.

2.6 Chip Power (VDD,VSS)

The MCS65E4 will by powered by +5.0 Volts DC applied between the VDD and VSS pins (VDD = +5, VSS = Ground).

2.7 Oscillator (Osc In, Osc Out)

The 8 Mhz oscillator can be controlled by a quartz crystal connected between the Oscillator In and Oscillator Out pins. In addition, the chip can be controlled by an external oscillator by driving the Oscillator In pin with a TTL level square wave.

2.8 Bus Clock (BCLK)

The Bus Clock corresponds to the normal Phase Two clock in the 6502 microprocessor system. Since this signal is always present, it can be used to synchronize the RDY, HOLD and BERR signals and to control data transfers between the MCS65E4 and any 6502 interface device.

2.9 Valid Memory Address

This bit goes high to indicate that there is a valid memory address on the address bus.

2.10 Memory Ready (RDY)

The Memory Ready input can be used to control the operation of the processor when interfacing to slower memory or peripheral devices. This signal operates in the same manner as in the 6502 microprocessor system with the additional capability of being able to stop on both a read and a write operation. These two operations are described separately below. The dynamic memory refresh operation is disabled as long as RDY is held low.

2.10.1 Operation of RDY during Read Cycle

At the beginning of a memory read operation, the processor places A9-A23 and the bus status information on the multiplexed address/data lines. This is followed by RAS going low to cause this information to be latched externally. The address and bus status information is then changed to A1-A8 and D0-D15. This is followed by CAS going low and BCLK going high.

Immediately after BCLK goes high, the RDY line can be pulled low to cause the processor to stop in its current state. If RDY is pulled low during a memory read operation, the processor stops with the data bus lines in the high impedance state. The RAS and CAS signals remain low as long as RDY remains low. This will hold the address in the external latches allowing whatever time is necessary for the memory outputs to become valid.

2.10.2 Operation of RDY during Write Cycle

Timing for the Write cycle is very similar to that described above for the Read cycle. The Write Enable Signals (WEL, WEH) will go low immediately after the beginning of the cycle (coincident with A9-A16 going valid). Immediately after RAS goes low, the data to be written into memory is placed on the DB0-DB15 lines. If RDY is pulled low during this cycle, the RAS and CAS signals remain low and the processor output data will remain on the DB0-DB15 lines. The Write Enable lines will go high coincident with the trailing edge of the BCLK pulse during which the RDY line returns high.

2.11 Interrupt Input (INT)

The MCS65E4 processor can be interrupted through the Interrupt Input. Setting the INT pin low causes the MCS65E4 to enter an interrupt sequence at the end of the current instruction if the Interrupt Inhibit bit in the Process Control Register is cleared. The operation of the interrupt function is described in detail in Section 4.0.

2.12 Reset (RES)

The processor can be reset by applying a low signal to this input. For power-on reset, this can be accomplished by connecting an R-C circuit to the RES pin. Positive control of the reset function in the peripheral devices can be accomplished by connecting these devices to the IORES Bus Status bit. As long as the reset input stays low, the processor will not perform any write operations.

2.13 Write Enables (WEL,WEH)

The write-enable signals control the direction of data transfers between the MCS65E4 and memory. If a write-enable line is high (Read), data will be transferred from memory to the processor. If this signal is low, data will be transferred into memory. WEL controls writing into the lower byte im memory (even addresses) while WEH controls writing into the the upper byte (odd addresses).

2.14 Bus Error

The Bus Error pin can be used to indicate that an error occurred during the previous cycle. This error can be the result of a Virtual Memory Address fault, a data error detected in an external EDC chip, or any other form of error. When this occurs, the processor immediately suspends its current execution sequence and traps to the operating system. The operating system can process the error and, if appropriate, can then return to the execution sequence which was interrupted.

2.15 Hold (HOLD)

The Hold pin can be pulled low to cause the processor to stop and to place its address and data bus into the high impedance state. This is used primarily for external DMA and multiprocessor operations. As long as the HOLD pin is low, the RAS and CAS signals continue to operate normally and the processor continues to put out the Bus Status bits. However, no address or data signals are generated by the processor and the corresponding pins remain in the high- impedance state except as required to generate the bus status information and to perform the required refresh operations. If the external memory refresh is enabled during the hold state (Hold = Low), the HOLDA bus status bit will return low periodically to signal the external devices that the processor will place refresh addresses on the address bus

2.16 Instruction Intercept (II)

The Instruction Intercept can be used to cancel the execution of an instruction within the MCS65E4. If this line is pulled low, the current instruction execution terminates immediately. The processor then treats the next byte in program sequence as an Op Code and immediately enters the appropriate execution sequence. This pin is used primarily by Auxiliary Arithmetic Processors to cancel the execution of intercepted instructions.

3.0 Internal architecture of the MCS65E4

3.1 Introduction

All aspects of the internal MCS65E4 architecture are desisned to achieve the desired level of performance in the smallest possible chip size. Most of the registers are organized into a single dynamic array with all data modification taking place in a high-speed 8-bit ALU. Four internal cycles are executed for each external (processor) cycle. This ratio of internal to external cycles combined with the fact that the ALU is utilized in nearly every internal cycle allows full 2 MHz operation in a processor containing a full 64 bytes of register within a chip size usually associated with 8 bit processors.

It should be noted that the device described below is only the first implementation of the architecture described in this document. This implementation tries to achieve a balance between chip size and capability with a strong emphasis on minimizing chip cost. It is assumed that future implementations of this architectute will result in devices with increased capability through larger control ROMS, through the integration of additional system functions (keyboard interface, etc.) onto the processor, and ultimately, by expanding the internal organization from 8 to 32 bits. All of these configurations will be upward compatible with the earlier devices.

The MCS65E4 is organized into an Execution Unit and an Execution Control Unit. Each of the major components which comprise these two units is described briefly below.

3.2 Execution Unit

3.2.1 ABL/ABM Registers

Those registers which are associated with the multiplexed low order sixteen address pins are located in a single dynamic array. These registers are:

1. Program Counter Low and Middle
2. Refresh Register
4. Address Register 1 Low and Middle
5. Address Register 2 Low and Middle
6. Address Register 3 Low and Middle

These registers are supported by an eight-bit incrementer which operates in parallel with the ALU described below.

3.2.2 Register Array

The complete register array is contained in a matrix of dynamic RAM cells. The traditional 3-2-2 dynamic RAM cell has been expanded to allow two READ buses and one WRITE bus. The register refresh operation is handled by a combination of hardware and software in a manner which is totally transparent to the user.

3.2.3 Arithmetic/Logic Unit (ALU)

Most of the data modification operations take place in the ALU. This includes normal execution operations as well as middle and high order Program Counter incrementing and register incrementing, decrementing, etc. The ALU is equipped with high speed carry look-ahead to allow it to complete any operation within one internal cycle. This allows an 8-bit ALU to perform most of the data manipulation functions required by a 32-bit processor.

The specific functions performed in the ALU are as follows:

1. Data shifting
2. Address limit checking
3. 2's complement binary addition and subtraction
4. Packed BCD addition and subtraction
5. Logic AND
6. Logic OR
7. Logic EOR

3.2.4 Input Queue

Data which is fetched from memory under control of the program counter is first loaded into the input queue where it is held until it is needed by the control logic. The queue is usually filled by 'pre-fetching' the next instruction sequence during each execution.

3.2.5 ABH/DB Registers

All of the registers associated with the Data Bus and the Address Bus High are located in a single dynamic array. This facilitates the multiplexing of these signals onto a set of sixteen pins as described in Section 2. These registers are as follows:

1. Program Counter High
2. Data Latch Low and High
3. Address Register 1 High
4. Address Register 2 High
5. Address Register 3 high

The bus status signals are generated in the control section and are multiplexed with the appropriate data bus signals at the bonding pad.

3.3 Execution Control Logic

3.3.1 Control Registers

All of the registers needed to assure proper instruction execution are contained in the Control Register Section. These registers perform such operations as storing execution control flags, selecting registers within the register array, counting execution cycles and addressing the microcode array.

3.3.2 Microcode Array

Both the microcode ROM and the Nanocode ROM are located in a single array. This assures minimum chip size since only one set of buses, decoder/drivers, etc. is required. In addition, this array is organized in a manner which allows the total size of the ROM to be varied without affecting the remainder of the chip. This will allow the rapid generation of additional versions of the processor which provide additional capability through expanded microcode.

4.0 MCS65E4 Microprocessor Software Architecture

4.1 Introduction

The primary goal of the MCS65E4 architecture is to shorten the gap between the processor hardware and the high level language architecture while at the same time retaining the generality which will allow it to support a broad range of applications. In particular, the software architecture of the MCS65E4 exhibits the following characteristics:

1. Strong multi-tasking support.
2. Separation between data and program.
3. 'Three-operand' addressing, i. e., all data operations are memory-to-memory.
4. Data structures (array, record, etc.) directly resembling high-level language practices.

4.2 MCS65E4 Internal Architecture

4.2.1 Introduction

The internal architecture of the MCS65E4 contains all of the registers needed to support execution of the instruction set described in Section 4.8. This set of registers is divided into those which are visible to the programmer (hereafter referred to as the Process Registers) and a set of temporary data registers which are used during instruction execution. The process registers are as follows:

1. Process Base Register (BAS)
2. Process Limit Register (LMT)
3. Process Program Counter (PPC)
4. Primary Base Register (PRM)
5. Top of Stack Register (TOS)
6. Process Control Register (PCR)

It should be noted that for speed purposes, the internal process registers (PRM, TOS, LMT, and PPC) contain physical addresses during process execution. However, the MCS65E4 user does not see these physical addresses since they are converted to logical addresses whenever the contents of one of these registers is transferred into memory. This is accomplished by subtracting the contents of the Process Base Register (BAS) from the address being transferred into memory. Similarly, the logical addresses contained in memory are converted to physical addresses when the internal process registers are loaded.

4.2.2 Process Base Register (BAS)

The Process Base Register sets the lower limit of the memory space in which the process must execute. This memory space starts at the first byte of the page whose address is contained in BAS, i.e., the BAS register contains the page number of the physical address at which the process starts. For example, if the BAS register contains 04E7, the lowest address which is available to the

process is 04E700.

4.2.3 Process Limit Register (LMT)

The Process Limit Register sets the upper limit of the memory space in which a process is to execute. At the same time, it identifies a page in memory which is used to store exception vectors and other information required to control execution of the process. The highest address available to a process is the last byte of the Limit Page. For example, if the LMT register contains 0D34, the highest available address in the process is 0D34FF.

4.2.4 Process Program Counter (PPC)

Execution of MCS65E4 programs proceeds under control of the Process Program Counter. The operations associated with this register are much the same as in any programmable processor.

4.2.5 Primary Base Register (PRM)

The Primary Base register is provided to control the accessing of data during instruction execution. Address offsets contained in the MCS65E4 instructions are added to the PRM register to obtain the physical address of the data.

4.2.6 Top of Stack Register (TOS)

The Top of Stack Register controls access to the process stack during instruction execution. The stack and Top of Stack Register (TOS) operate in a conventional manner to store subroutine return addresses, subroutine data, interrupt return addresses, etc. In addition, the TOS can be used as a base register to control the accessing of data in memory utilizing offset addressing. This operates in exactly the same manner as does Offset Addressing using the Primary Base (PRM) register.

4.2.7 Process Control Register (PCR)

The Process Control Register contains a number of flags and control bits which are used to control instruction execution within the processor. The PCR register bits are:

Bit	Designation
0	K - Kernel Mode Flag
1	U - User/Supervisor Mode
2	I - Interrupt Inhibit Flag
3	E - Enable External Memory Refresh
4	P - Enable Periodic Interrupt
5	S - Enable Stack Boundary Check
6	D - Debug Mode
7	T - Disable All Traps
8-11	M - Microcode Select
12-15	R - Refresh Rate

4.2.7.1 Kernel Mode Flag (K)

The Kernel state (K = 1) represents the first level of the operating system. This flag is set and cleared automatically as the processor moves into and out of the Kernel state.

4.2.7.2 User/Supervisor Mode (U)

The User/Supervisor flag is set to a logic 1 to enable execution of a number of privileged instructions which are normally available only to the operating system. This flag is set automatically by the Reset input or when the processor exits from a User process. It is cleared when a User process is invoked.

4.2.7.3 Interrupt Inhibit Flag (I)

The Interrupt Inhibit Flag can be set to disable interrupts on the INT input.

4.2.7.4 Enable External Memory Refresh (E)

The E flag must be set to a logic 1 to enable the processor to perform periodic external memory refresh operations. The internal refresh logic will assure that each row in the dynamic memories will be refreshed at a rate determined by the Programmable Refresh Control Counter.

4.2.7.5 Enable Periodic Interrupt (P)

The P flag can be set to a logic 1 to cause the processor to execute a trap each time the memory refresh logic 'rolls over'. This occurs at a rate determined by the Refresh Control Counter (typically between 2 and 4 milliseconds). This trap will occur whether or not the external refresh operation is enabled.

4.2.7.6 Enable Stack Boundary Check (S)

The S flag can be set to cause the processor to execute a trap whenever the stack crosses a page boundary during a PUSH or POP operation. This allows either the process or the operating system to verify that the stack will not over-write data in memory.

4.2.7.4 Debug Mode (D)

The Debug flag can be set to allow single-instruction execution of a User process. Each time the processor enters a User process it will execute a single instruction and will then trap back to the operating system, allowing the operating system to display the effects of each instruction execution for debug purposes.

4.2.7.8 Enable Read Before Byte Write (W)

All traps are disabled if this flag is set.

4.2.7.9 Microcode Select (M)

These four bits directly reflect the contents of the internal Microcode Select Register. This data is placed onto bits 12-15 of the address bus during an external microcode fetch.

4.2.7.10 Refresh Rate

These four bits directly reflect the contents of the internal Refresh Control Register. This data directly controls the rate at which the MCS65E4 refreshes the external memories.

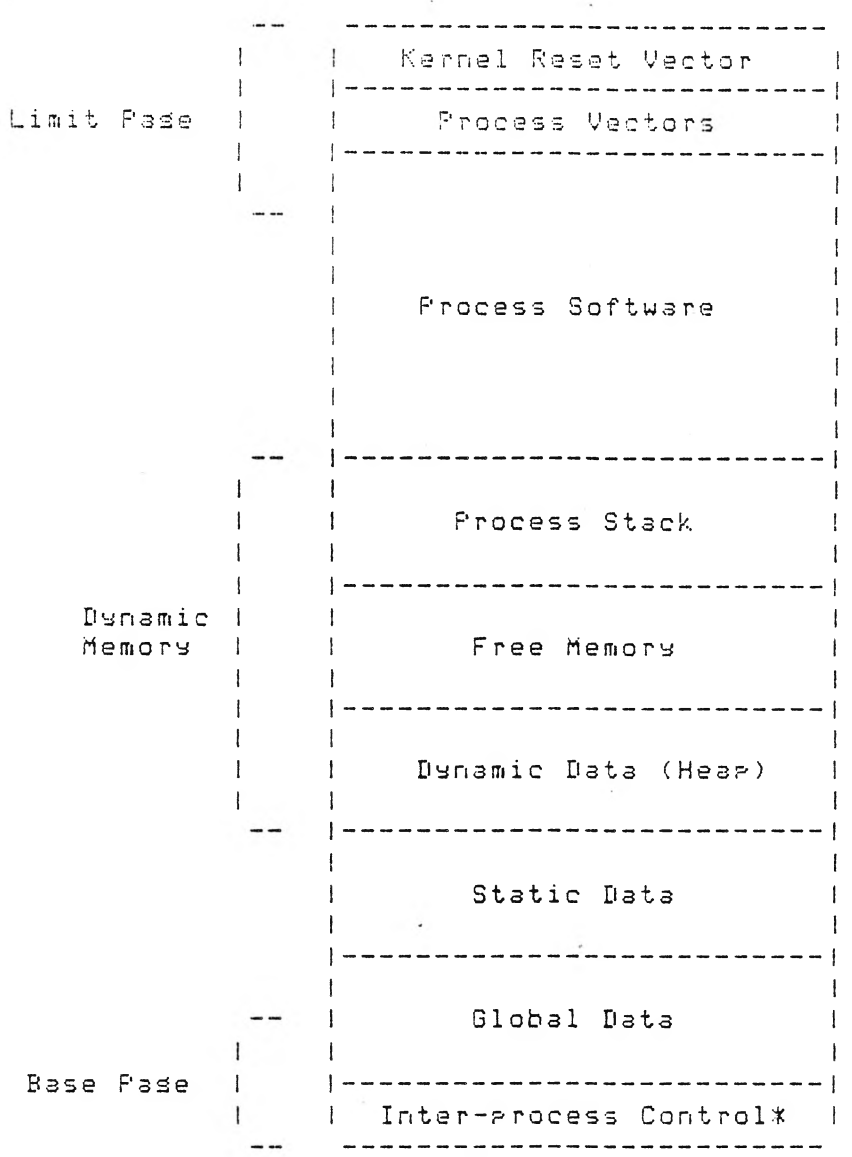


Figure 4.0 Suggested Process Organization in the MCS65E4 System

4.3 Process Structure

4.3.1 Introduction

Those factors which influence the organization of a process within the MCS65E4 are much the same as the factors which govern the organization of memory within an MCS6502 system. These are as follows:

1. The vectors associated with the processing of interrupts, system calls, etc. are located in the Limit page, i. e., in high order memory. This generally dictates that program memory be at the upper limit of the address space allocated to the process.

2. The availability of short offsets from the Process Base Register would seem to dictate that Read/Write memory be located in the low order portion of the address space allocated to a process. In addition, the first three bytes of memory within the Kernel process and within any Operating System process must be read/write memory.

These factors lead to the general process organization shown in Figure 4.0. However, this process structure is by no means mandatory. This is particularly true if the entire process is located in read/write memory. As long as the process vectors remain in the addresses outlined, it is possible to place the process software anywhere in the process address space, such as directly above the static data area. This would place the entire dynamic data area (including the stack) in high order memory.

4.3.2 Inter-process Control

As mentioned above, the first three bytes of the Kernel and Operating System processes must be reserved for use by the MCS65E4 to control movement into and out of the process. These addresses must be located in read/write memory. The processor will transfer data into and out of this area during the servicing of interrupts, system calls, etc.

The MCS65E4 architecture does not require that the first three bytes of the User process be reserved.

4.3.3 Global Data

The first 64 bytes of memory above the process base can be accessed with a single byte of addressing information. In addition, addresses 65 through 511 can be accessed with one additional byte of offset (two bytes total). For this reason, this region should be used to store those static variables which are accessed most frequently.

4.3.4 Static Data

This area consists primarily of static variables which will be utilized by the process software. This data should be accessed

through the Primary Base Register or through the Process Base Register.

4.3.5 Dynamic Memory

The dynamic memory area consists of three sections. These are the dynamic data area, the free memory area and the process stack. Each of these is discussed separately below.

4.3.5.1 Dynamic Data

The first section is the dynamic data area in which the processor allocates memory to dynamic variables during process execution. This data can be accessed through the Primary Base Register and through the external base registers. This area may be used for a process heap, or for the storage of higher level processes called during execution of process software.

4.3.5.2 Free Memory

The free memory area acts as a buffer between the dynamic data and the process stack. Since the MCS65E4 stack grows downward toward lower-order memory, the optimum configuration would be that in which the dynamic data area grows upward into the free memory area while the stack grows downward. The MCS65E4 architecture contains provision for assuring that these two data areas do not overlap.

4.3.5.3 Process Stack

Transfer of data into and out of the process stack is performed under control of the Top of Stack Register within the MCS65E4 processor. In the MCS65E4, the TOS register always contains the physical address of the last byte of data placed onto the stack. Therefore, the TOS register is decremented before data is placed into the stack and is incremented after each transfer of data out of the stack.

4.3.6 Process Software

The process software can generally be viewed as 'static', i.e., the memory requirements will not change during the execution of the process. Therefore, this software should generally be located outside of the dynamic memory area. As outlined above, MCS65E4 architecture requires that a set of process vectors be located in fixed positions within the process address space. For this reason, it will generally be more satisfactory to place the process software in high-order memory along with these vectors.

4.3.7 Process Vectors

Processing of interrupts, system calls, and system errors is controlled by a set of vectors which must be located in the Limit Page of the process.

4.3.8 Kernel Reset Vector

In the Kernel Process, the high order four bytes of memory are reserved for storing the Kernel Reset Vector. This is used by the processor during the system reset operation. The Reset Vector is stored in the same format as the exception vectors.

4.4 Execution of Processes in the MCS65E4

4.4.1 Introduction

The registers described in Section 5.2.1 are designed to support the execution of a hierarchy of processes in a multi-task environment under the control of a sophisticated operating system. One of the key aspects of this architecture is support from the processor to initiate a new process, to exit from a process in the event of a fault or interrupt, and to return to an interrupted process. All of these inter-process operations are described in this section.

4.4.2 Basic Inter-process controls

4.4.2.1 Introduction

The MCS65E4 provides five primary tools for controlling movement into and out of processes. They are the following:

1. Kernel Reset Vector
2. Process Parameter List
3. Process Link
4. Pointer to Current Caller
5. Process Stack

4.4.2.2 Kernel Reset Vector

The high order four memory locations in the Kernel process (physical addresses FFFFFC-FFFFFF) are reserved for storage of the Reset vector.

4.4.2.3 Process Parameter List (PPL)

The Process Parameter List (PPL) contains the information necessary to enter a process for the first time. The arguments in this list are as follows:

1. List Size

   This eight-bit parameter specifies the number of bytes of data contained in the list (not including the List Size parameter).

2. Process Base Address

   The Process Base Address parameter specifies the logical page address of the Base Page of the new process within the caller's address space.

3. Process Size

This 16-bit parameter specifies the logical page address of the Limit Page within the new process, i. e., the page address relative to the new process's base, not to the caller's base. This data is used by the MCS65E4 to load the Process Limit Register during process initialization.

4. Program Entry Address

This 24-bit parameter specifies the logical address within the new process of the entry point for the process software. This data is used to load the Process Program Counter during process initialization.

5. PRM Initial Value

This 24-bit parameter specifies the logical address within the new process of the initial Primary Base Register contents. This data is used to load the PRM register during process initialization.

6. TOS Initial Value

This 24-bit parameter specifies the logical address within the new process of the initial top of stack. The MCS65E4 uses this data to load the TOS register during process initialization.

7. Process Control Register Initial Value.

This 16-bit parameter specifies the initial contents of the Process Control Register. This data is transferred directly into the PCR during process initialization.

4.4.2.4 Pointer to Current Caller

During the execution of any process (other than the Kernel process) it is very important that the MCS65E4 be able to exit from the process and return to its caller. This is accomplished by utilizing physical addresses 000000 through 000002 within the Kernel process to store the physical address of the current caller's Top of Stack. This information will be utilized by the MCS65E4 during the processing of any exceptions which require that execution of the current process be suspended.

Addresses 000000-000002 within the Kernal Process are reserved and should not be used by the Kernel software for general data storage.

4.4.2.5 Process Link

For Operating System processes it is necessary that the MCS65E4 be able to exit to both lower level and higher level processes. The Pointer to Current Caller described previously stores the physical address of the caller's top of stack, allowing a process to return to its caller at any time. However, when a higher level process is

invoked, it will be necessary to store this pointer in a manner which assures that it will be available when the MCS65E4 exits from the higher level process and begins executing the intermediate level process once again. This is accomplished by storing the information contained in the Pointer to Current Caller into the first three bytes of the intermediate process (logical addresses 000000 through 000002) before exiting to the higher level process. These three logical addresses will be referred to as the Process Link.

It should be noted that the Process Link is a reserved area and should not be used by the Operating System Process for general data storage. In addition, this process link does not exist within the User process since it is impossible to invoke additional processes within the User process.

4.4.2.6 Process Stack

Throughout the inter-process operations described below, the process stack is utilized for saving the internal registers when exiting from a process.

4.4.3 Inter-Process Operations

4.4.3.1 Introduction

The manner in which each of these architectural elements is used in a system can be described most effectively through a detailed discussion of the primary inter-process operations that must take place during the operation of a full scale multi-task microcomputer. Specifically, these operations are:

1. System Reset.
2. Invoking higher level processes.
3. Exiting from a process in the event of an interrupt, system call, or bus error.
4. Returning to a process after an interrupt, system call, or bus error.

4.4.3.2 System Reset

When the MCS65E4 is reset, it immediately enters the Kernal mode with the Base, Limit and Process Control registers initialized as follows:

Register	Initial Contents
Process Base (BAS) Register	0000XX
Process Limit (LMT) Register	FFFFXX
Process Control Register (PCR)	
Bit 0 (K)	1
Bit 1 (U)	1
Bit 2 (I)	0

Bit 3	(E)	0
Bit 4	(P)	0
Bit 5	(S)	0
Bit 6	(D)	0
Bit 7	(W)	0
Bits 8-11	(M)	0
Bits 12-15	(R)	0 (2 MSEC Refresh rate)

The processor then fetches a 24-bit address utilizing the information stored in the Reset Vector (addresses FFFFFFC through FFFFFFF) and begins executing the instructions located at this address. The manner in which the Reset Vector information is utilized to determine the Kernel starting address is described in Paragraph 4.7.6.2.

4.4.3.3 Invoking Additional Processes

Higher level processes can be invoked either from the Kernel process or from an Operating System process by executing an IOS or TASK instructon. The single operand contained in the instruction must point to the Process Parameter List for the new process. The MCS65E4 begins execution of the IOS or TASK instruction by placing the contents of the internal registers onto the current process stack and then examining the Process Parameter List to determine the operating parameters for the new process. More specifically, the sequence proceeds as follows:

1. The contents of the Process registers are pushed onto the current process stack. After this operation, the process stack contains the following:

Memory Location	Contents
TOS+11	PCR, Bits 8-15
TOS+10	PCR, Bits 0-7
TOS+9	PRM, Bits 16-23
TOS+8	PRM, Bits 8-15
TOS+7	PRM, Bits 0-7
TOS+6	PPC, Bits 16-23
TOS+5	PPC, Bits 8-15
TOS+4	PPC, Bits 0-7
TOS+3	LMT, Bits 8-15
TOS+2	LMT, Bits 0-7
TOS+1	BAS, Bits 8-15
TOS	BAS, Bits 0-7

2. The processor then fetches the contents of the Pointer to Current Caller (physical addresses 000000-000002) and places this information into the Process Link of the current process (logical addresses 000000-000002 of the current process).

3. The Pointer to Current Caller is then updated by transferring the data contained in the TOS register into physical addresses 000000 through 000002.

There are several items worth noting in the preceeding operations. The first is that the addresses which are placed onto the caller's stack from the internal process registers are physical addresses. No attempt is made to convert these physical addresses to logical addresses. In addition, the processor does not attempt to format this data in a manner which would facilitate subsequent manipulation through the normal processor software. Both of these are made possible by the basic nature of process execution within the MCS65E4 architecture. In particular, it is assumed that the limits within which the new process will execute will not include that portion of memory in which the caller's stack is located and that it will not include the caller's Base Page or Limit Page. Therefore, it will be impossible for any higher level process to access this data. Similarly, the use of physical addresses on the caller's stack is made possible by the fact that the caller cannot be moved within the physical address space while a higher level process is being executed.

After the preceeding operations are complete, the MCS65E4 is ready to enter the new process. This is accomplished as follows:

1. The processor first calculates the physical address of the PPL for the new process. This is accomplished through one of the normal operand addressing sequences described in Section 4.5 utilizing the information contained in the instruction. This physical address is then transferred into one of the internal registers for use during the remainder of this operation.

2. The first item in the PPL specifies the number of bytes of data which are contained in the list. This list length parameter is transferred into an internal register to control termination of the IOS or TASK instruction.

3. The second parameter in the PPL contains the logical address of the base of the new process. This is expressed as a 16-bit logical page address within the calling process. The physical address of the base of the new process is determined by adding this data to the caller's physical base address. The resulting physical page address is loaded into the Process Base register.

4. The third item in the Process Parameter List specifies the number of pages of memory which must be allocated to the process. This information is added to the Process Base Register. The resulting 16-bit physical page address is transferred into the Process Limit Register.

5. The fourth item in the PPL specifies the logical address of the process entry point, i. e., the "reset vector" for the process. This 24-bit parameter is added to the contents of the Process Base Register, i. e. to the base address of the new process, to determine the physical address of the process entry point. This physical address is loaded into the Process Program Counter.

Those items listed above represent a minimum length PPL. However, it is possible to initialize additional process registers by extending the length of the PPL. In this case, the MCS65E4 will fetch additional parameters in the following order:

1. The first parameter in the extended PPL specifies the logical address of the initial Top of Stack within the new process. This data is added to the Process Base Register. The resulting physical address is transferred into the TOS register.

2. The next parameter in the PPL specifies the initial contents of the Process Control Register. This information is transferred directly into the PCR.

3. The next parameter specifies the initial value of the Primary Base Register. This is expressed as a 24-bit logical address within the new process. This data is added to the contents of the Process Base Register. The resulting physical address is transferred into the Primary Base register.

4.4.3.4 Exception Processing

4.4.3.4.1 Introduction

Exiting from the current software occurs whenever the MCS65E4 encounters one of a number of exceptions. These exceptions may be the result of a signal on one of the processor's input pins (interrupt, etc.), it may be the direct result of process software (System Call instruction, Data Access Trap, etc.), or it may be due to a problem in the data being processed. Any of these conditions will cause the MCS65E4 to discontinue execution of the current software and to begin execution of an exception handler. The procedures involved in doing so are described in detail in this section.

The MCS65E4 exceptions can be divided into a number of classes. The first are those "privileged" exceptions which must be serviced by the operating system (by a Supervisor Mode process). The second consists of those which can be serviced within a User process. In addition, a number of exceptions are recognized at the end of an instruction execution sequence while others must be recognized and processed immediately. As shown in the discussion below, each of these exception groups is handled differently.

All exceptions, regardless of type, are serviced under control of an exception vector located in a reserved portion of the Limit Page. Within the User process, however, there is no provision for storing vectors for the privileged exceptions. These exceptions are serviced by returning immediately to the current process's caller. For this reason, the User process vectors are a subset of those which must be stored in the Supervisor Mode processes (Operating System and Kernel). All of these exceptions are described in detail in Section 4.4.4. The Exception Vector format and the manner in which the MCS65E4 utilizes the exception vector

information to determine the address of the exception handler are described in Section 4.7.6.

4.4.3.4.2 Servicing Exceptions Within the Current Process

Many of the fault conditions occuring within the MCS65E4 system can be serviced by the currently executing process. When a non-privelesed exception occurs during the execution of a User Process or when any exception occurs during execution of an Operating System process, the MCS65E4 immediately checks the TRAP bit in the appropriate exception vector. If this bit is a logical zero, the exception is serviced in the current process as described in this paragraph. If the TRAP bit is a logic 1, the processor will return to the current caller for servicing the exception. (Note that all exceptions which occur during execution of the Kernel Process must be serviced within the Kernel Process software.) The sequence of operations which takes place when an exception is serviced within the current process depends on whether the exception is recognized at the end of each machine cycle or between instructions. For those exceptions which are recognized only between instructions, the sequence proceeds as follows:

1. The physical address contained in the Process Program Counter is converted into the corresponding logical address and is then pushed onto the current process stack. The TOS Register is adjusted to point to the last byte of data which was pushed onto the stack. After the above sequence is complete, the process stack contains the following:

Memory Location	Contents
TOS+2	PPC, Bits 16-23
TOS+1	PPC, Bits 8-15
TOS	PPC, Bits 0-7

2. The MCS65E4 then loads the physical address of the exception servicing software into the Process Program Counter. This address is determined by adding the logical address referenced by the exception vector to the contents of the Process Base Register. The processor then begins executing the software at this address.

After servicing of the exception is complete, the processor can return to the process software at the point where the exception occurred by executing a Return From Subroutine (RTS) instruction. This transfers data from the process stack into the Process Program Counter, adjusting the TOS Register to point to the last valid byte of data in the stack.

For those exceptions which must be recognized at the end of a processor cycle rather than at the end of the current instruction, the processing sequence is much more complex than that just

described. This is caused by the fact that it will be necessary to return to the middle of an instruction execution sequence after exception processing is complete. This is accomplished by placing the contents of all of the internal registers onto the stack, including all of the Process registers (except the Process Base Register and the Process Limit Register), all of the temporary data registers used during instruction execution and all of the various latches, registers, etc. which control the instruction execution sequences. This sequence of operations proceeds as follows:

1. The MCS65E4 first places the contents of the Process Registers, the temporary data registers, and the misc. execution control registers, latches, etc. onto the stack. The processor does not attempt to organize this data in a manner which will facilitate processing of this information by the exception handler. After this operation is complete, the process stack contains the following:

Memory Location	Contents
TOS+59	Temporary Operand Register 8
TOS+58	. . .
TOS+57	. . .
TOS+56	. . .
TOS+55	Temporary Operand Register 7
TOS+54	. . .
TOS+53	. . .
TOS+52	. . .
TOS+51	Temporary Operand Register 6
TOS+50	. . .
TOS+49	. . .
TOS+48	. . .
TOS+47	Temporary Operand Register 5
TOS+46	. . .
TOS+45	. . .
TOS+44	. . .
TOS+43	Temporary Operand Register 4
TOS+42	. . .
TOS+41	. . .
TOS+40	. . .
TOS+39	Temporary Operand Register 3
TOS+38	. . .
TOS+37	. . .
TOS+36	. . .
TOS+35	Temporary Operand Register 2
TOS+34	. . .
TOS+33	. . .
TOS+32	. . .
TOS+31	Temporary Operand Register 1
TOS+30	. . .
TOS+29	. . .
TOS+28	. . .
TOS+27	Temporary Address Registers

```
TOS+26          '           '           '
TOS+25          '           '           '
TOS+24          '           '           '
TOS+23          '           '           '
TOS+22          '           '           '
TOS+21          '           '           '
TOS+20          '           '           '
TOS+19          '           '           '
TOS+18      PRM, Bits 16-23
TOS+17       ' , Bits 8-15
TOS+16       ' , Bits 0-7
TOS+15      PPC, Bits 16-23
TOS+14       ' , Bits 8-15
TOS+13       ' , Bits 0-7
TOS+12      Literal Register
TOS+11          '           '
TOS+10      Internal Control Registers
TOS+9           '           '           '
TOS+8           '           '           '
TOS+7           '           '           '
TOS+6           '           '           '
TOS+5           '           '           '
TOS+4           '           '           '
TOS+3       Data Input Latch, Bits 8-15
TOS+2                '          , Bits 0-7
TOS+1       Data Output Latch, Bits 8-15
TOS                  '          , Bits 0-7
```

2. The MCS65E4 then loads the physical address of the exception servicing software into the Process Program Counter. This address is determined by adding the logical address referenced by the exception vector to the contents of the Process Base Register. The processor then begins executing the software at this address.

After servicing of the exception is complete, the processor can return to the process software at the point where the exception occurred by executing a Return From Exception (RTE) instruction. This transfers all of the data previously placed onto the process stack back into the appropriate processor registers. The TOS Register is then adjusted to point to the last valid byte of data in the stack. The MCS65E4 then begins execution of the instruction sequences which had been interrupted by the exception.

4.4.3.4.3 Servicing Exceptions Within the Current Caller

If the TRAP bit is set within the exception vector, or if a privileged exception occurs within a User Process, the MCS65E4 must exit from the current process and return to the current process's caller to service the exception. This sequence of operations is much more complex than that described above because of the need to push the contents of all of the process registers onto the stack to allow a return to the process at a later time. The sequence of operations which takes place depends primarily on whether the exception was recognized at the end of an instruction

execution or during an instruction. In the latter case, the processor will begin the exception processing by first placing the contents of all of the internal registers in exactly the same sequence as described above (Paragraph 4.4.3.4.2). After this operation is complete, the remainder of the Process Registers are placed onto the stack in a manner which is compatible with the Process Parameter List described above. This allows the MCS65E4 to return to the process at a later time utilizing the same IOS or TASK instruction which was used to enter the process initially.

After this sequence of operations is complete, the processor is ready to exit from the process and to return to the current caller. This is accomplished as follows:

1. The processor first fetches the address of the current caller's top of stack from absolute addresses 000000-000002. This is placed into the TOS Register.

2. If the processor is returning to an Operating System process, the data in the Process Link is then moved into the Pointer to Current Caller.

3. The Processor next transfers the data from the caller's stack into the internal process registers. This allows the processor to begin executing the caller's exception servicing software.

4. Before executing the exception handler within the caller, the processor first pushes the information which will allow returning to the process which was interrupted by the exception. This is accomplished by first calculating the logical address within the caller of the Top of Stack for the interrupted process by subtracting the physical address of the Caller's base from the physical address of the Top of Stack for the higher level process. This information is then pushed onto the caller's stack.

5. If there are any exception qualifiers associated with the exception, this information is then pushed onto the caller's stack.

6. The processor then fetches the appropriate exception vector from the Limit Page of the caller. The logical address referenced by this vector is transferred into the Process Program Counter. The MCS65E4 then begins executing the software located at this address.

4.4.3.5 Returning to a process after an interrupt, system call, or bus error.

The procedures which allow returning to a process which had been suspended by the occurrance of an exception are exactly the same as those which are utilized for entering the process for the first time. This is made possible by the fact that the processor created a Process Parameter List on the stack of the suspended process before exiting. When returning to the process, the single operand

in the IOS or TASK instruction must reference this PPL. This is facilitated by the fact that the logical address of this PPL within the caller had been placed onto the caller's stack. This logical address can be accessed by referencing the Top of Stack within the TASK or IOS instruction. Note that the amount of data which must be transferred into the internal processor registers is controlled by the List Length parameter which was placed on the stack last. This allows the same procedures for returning to the process whether the exception was recognized after an instruction execution sequence or in the middle of an instruction.

4.4.4 Exception Vectors within the MCS65E4 Process.

4.4.4.1 Introduction

The Exceptions Vectors and their location within the limit page are as follows:

Non-Privileged Exceptions	Address
1. Undefined Op Code	F8-FB
2. Undefined data type	F4-F7
3. Subscript out-of-limits	F0-F3
4. Operator and Operand not compatible	EC-EF
5. Overflow	E8-EB
6. Other arithmetic error (divide by zero, etc.)	E4-E7
7. Non-conformable data types	E0-E3
8. Instruction Access Trap	DC-DF
9. Data Access Trap	D8-DB
10. Stack Page Boundary Trap	D4-D7

Privileged Exceptions	Address
1. Interrupt Request (IRQ)	D0-D3
2. System Call	CC-CF
3. System Call with message	C8-CB
4. Channel Trap, Channel A	C4-C7
5. Channel Trap, Channel B	C0-C3
6. Bus Error	BC-BF
7. Access out-of-limit	B8-BB
8. Debug Trap	B4-B7

Each of these is described in detail below. Note that within the User process, addresses D4 through FB are reserved for the storage of exception vectors. Addresses B4 through D3 are available for general data storage. In the Supervisor Mode processes, addresses A8 through FB are reserved and should not be used for general data storage. Addresses FC through FF are used within the Kernel process to store the Kernel Reset Vector. Addresses FC-FF cannot be used for general data storage in either the Supervisor or User Modes.

The exception vectors are 24-bit pointers stored in memory in a format which is compatible with the data descriptor described in Section 4.7. Within the exception vector descriptor header, the TRAP bit is used to indicate when the exception is to be serviced

in the current process. If this bit is a logic zero, the vector is assumed to reference a logical address within the current process. The format of the Exception Vectors and the manner in which the the MCS65E4 utilizes the information in the Exception Vector to determine the address of the exception handler are described in Section 4.7.6.

4.4.4.2 Undefined Op Code Trap

The MCS65E4 will trap through this vector whenever it encounters an Op Code which is not supported in the standard instruction set. This allows the process to either abort or to interpret the Op Code through the exception processing software.

4.4.4.3 Undefined Data Type Trap

The MCS65E4 will trap through this vector when it encounters a data type which is not supported in the standard instruction set. This allows the process to abort or to interpret the data type information in the exception processing software.

4.4.4.4 Subscript out-of-limits Trap

The MCS65E4 will trap through this vector when the software attempts to access a structure or array with a subscript which is out of the limits specified in the data descriptor (see Section 4.7.4.3).

4.4.4.5 Operator and Operand not Compatible

The MCS65E4 will trap through this vector when the software attempts to perform an operation on a data field which is not compatible with the operation. This allows the process to either abort or to interpret the operation in the exception servicing software.

4.4.4.6 Overflow

The MCS65E4 will trap through this vector when an arithmetic overflow is encountered during execution of an arithmetic instruction.

4.4.4.7 Other arithmetic error (divide by zero, etc.)

The MCS65E4 will trap through this vector when it encounters any arithmetic error other than overflow. Specifically, these errors are the following:

1. Divide-by-zero.
2. Square root of negative number.

4.4.4.8 Non-conformable data types

The MCS65E4 will trap through this vector when the software attempts to perform an operation on two incompatible data fields. This allows the process to abort or to perform an automatic

conversion of one of the operands.

4.4.4.9 Instruction Access Trap

The MCS65E4 will trap through this vector when it begins execution of an instruction in which the TRAP bit is set.

4.4.4.10 Data Access Trap

The MCS65E4 will trap through this vector when it encounters a descriptor with the TRAP bit set to a logic 1.

4.4.4.11 Process Stack Page Boundary Trap

The MCS65E4 will trap through this vector when the Top-of-Stack Register crosses a page boundary during a Push or Pop operation. This allows the process or the operating system to verify that the process stack has not over-written other data in the dynamic data area.

4.4.4.12 Debug Trap

The MCS65E4 traps through this vector whenever it enters a User process with the Debug flag set. This exception can only be serviced in a Supervisor Mode process.

4.4.4.13 Interrupt Input (INT)

The MCS65E4 will trap through this vector when the Interrupt Input goes low during process execution.

4.4.4.14 System Call

The MCS65E4 will trap through this vector when it encounters an SC instruction during the execution af a process.

4.4.4.15 System Call with Message

The MCS65E4 will trap through this vector when it encounters an SCM instruction during the execution af a process.

4.4.4.16 Bus Error

The MCS65E4 will trap through this vector when the Bus Error input signal goes low during instruction execution. This operation is described in detail in Paragraph 4.4.3.4.2.

4.4.4.17 Access out-of-limit

The MCS65E4 will trap through this vector when the processor software attempts to access a physical address which is outside of the limits specified by the Process Base Register and the Process Limit Register, i. e., above address FF in the Limit page or below address 00 on the Base Page.

4.5 Addressing within the MCS65E4

4.3.1 Introduction

There are three primary locations in which data can be stored within the MCS65E4 architecture. These are:

1. In the internal processor registers.
2. In the instructions which form the process software.
3. In the data storage area of the process address space.

The MCS65E4 operand structure contains provisions for referencing data in each of these locations. Referencing data in the internal registers is accomplished by specifying the register in the first byte of the operand. Immediate data can be specified either in the first byte of the operand (short form immediate addressing) or in the operand extension bytes (long form immediate addressing). All other data references are accomplished by adding the value of an offset contained in the instruction to the contents of a base register. This base register can be either one of the internal registers or any three-byte location in the Base Page of the process.

The most important advantages associated with the use of base registers to control the accessing of data is that it assures complete relocatibility for both the process and for the data within a process and, in addition, it facilitates the creation and manipulation of dynamic data. This recognizes that most software routines manipulate a relatively small amount of data. Very seldom does a program find it necessary to access the entire 24-bit address space while manipulating data. For this reason, it should be possible to access most of the data with one or two bytes of offset instead of the three bytes which would be necessary if conventional absolute addressing were the only available addressing mode. Utilizing the base registers within the MCS65E4 in an effective manner can result in a significant reduction in the total program storage requirements by reducing the amount of addressing information which must be provided.

The first byte of each operand specifies the following:

1. The internal register which contains the data (register addressing.)

2. The value of the immediate data (Short Form Immediate Addressing).

3. The number of extension bytes of immediate data which follow (Long Form Immediate Addressing).

4. The base register to be used (internal or external).

5. The offset between the base and the desired data (Short Form Offset Addressing).

6. The number of bytes of offset which follow (Long Form Offset Addressing).

7. The Variable Access Mode (Byte, Two-byte Integer, Ordinal, or data field defined by a descriptor).

The data contained in the first byte of the Operand (hereafter referred to as the 'Operand Control Byte') can be organized into a number of fields and sub-fields. This organization is summarized in Figure 4.1 below. Each of the divisions in this figure are described in detail in subsequent paragraphs of Section 4.

BIT 7	6	5	4	3	2	1	0	Remarks
0	0	Number of Extension Bytes		Base Register Select		Data Access Format		Primary Addressing Group
0	1	0	0	Addressing Mode Select		Auxiliary Data Field		Secondary Addressing Group
0	1	0	1	Register				Internal Register Addressing
0	1	1	Immediate Data					Immediate Data Short Form
1	0	Offset from BAS						BAS Offset Addressing Short Form
1	1	Offset from PRM						PRM Offset Addressing Short Form

Figure 4.1. Organization of Operand Control Byte

4.5.2 Primary Addressing Group

4.5.2.1 Introduction

The Process Base Register (BAS), Primary Base Register (PRM), and Top of Stack Register (TOS) are the principal addressing registers within the MCS65E4. In addition, any three consecutive bytes of memory within the Process Base Page can serve as a base register during data accessing operations. For this reason, the Operand Control Byte is organized in a manner which assures that each of these registers can be used with maximum effectiveness during process execution. This is accomplished through the primary addressing group and by providing short form addressing for the BAS and PRM registers. This short form addressing is described in subsequent paragraphs of Section 5.6. Long form addressing for these internal and external base registers is described in this paragraph.

The addressing information provided in this group can be divided into three fields. The first group specifies which of the registers is to be utilized as the base register. The second specifies the number of bytes of addressing information which follows the Addressing Control Byte, and the third specifies the format of the data acquisttion. Each of these is described in detail below.

4.5.2.2 Base Register Select Field

The Base Register Select Field specifies one of the base registers as follows:

Bit 3	Bit 2	Selected Register
0	0	Process Base Register (BAS)
0	1	Primary Base Register (PRM)
1	0	Top of Stack (TOS)
1	1	External Base (EXTXX)

As described previously, the physical address of the data which is to accessed is determined by adding the offset contained in the instruction to the register selected by this field. If External Base Addressing is selected, the address of this base register (within the Process Base Page) is specified by the byte following the Operand Control Byte.

4.5.2.3 Data Access Format

The two bits of the Data Access Format field are used to control the manner in which data is to be accessed. This is accomplished as follows:

Bit 1	Bit 0	Format
0	0	Descriptor Access Data is to be accessed through a descriptor.

The address in the instruction is assumed to be that of a descriptor which contains all of the information required to properly manipulate the desired data field.

0 1 Single Byte Access

The address contained in the instruction is assumed to be that of a single byte of unsigned integer data.

1 0 Two byte Integer Access

The address contained in the instruction is assumed to be that of a sixteen-bit word of 2's complement integer data. Low order data is assumed to be located in the low order address.

1 1 Three byte ordinal Access

The address contained in the instruction is assumed to be that of a 24 bit ordinal data field. Low order data is assumed to be in the low order address.

4.5.2.4 Number of Extension Bytes

The Number of Extension Bytes field specifies the number of bytes of offset information which follows the Operand Control Byte. This is specified as follows:

Bit 3	Bit 2	Number of Addressing Bytes
0	0	None- The offset address is assumed to be zero.
0	1	One- The high order bits of the offset are assumed to be zeros. (offset value : 0 <= offset <= 255)
1	0	One- Bit 8 of the offset is assumed to be a logic 1. The remaining high order bits of the offset are assumed to be zeros. (offset value : 256 <= offset <= 511)
1	1	Two- The low order 8 address bits of the offset follow the Operand Control Byte. This is followed by bits 8-15. The high order eight address bits are assumed to be zeros. (offset value : 0 <= offset <= 655365)

4.5.3 Secondary Addressing Group

4.5.3.1 Introduction

In addition to these primary addressing modes, there are several addressing modes within the MCS65E4 software architecture which do not require the flexibility which is inherent in the addressing described in the previous paragraph. This is true for Limit Page addressing (utilizing the LMT Register as base) since this operation never requires more than a single byte of offset. Similarly, the PUSH and POP operations do not require any addressing information since the data is placed directly onto the process stack without offset. Finally, the Immediate Addressing requires only that the size of the immediate operand be specified since the data follows directly after the Operand Control Byte. These addressing modes are selected by the Addressing Mode Select field as Follows:

Bit 3	Bit 2	Addressing Mode
0	0	Limit Page Addressing
0	1	Process Stack PUSH / POP
1	0	Immediate Addressing (Long Form)
1	1	Process Base Addressing (Long form)

Each of these modes is described in detail below.

4.5.3.2 Limit Page Addressing

As described previously, the Limit Page within a process is used to store the vectors which are used in the servicing of interrupts, system calls, etc. during process execution. These vectors can be manipulated directly by the process whenever appropriate. Limit Page addressing provides an efficient method of accessing these vectors. When Limit Page addressing is selected, the Auxiliary Data Field (bits 0 and 1) specifies the Data Access Format in exactly the same manner as that described above (See Paragraph 5.6.2.3).

4.5.3.3 Process Stack PUSH / POP

The top of the process stack can be specified as the source (POP) or destination (PUSH) for data within most of the MCS65E4 instructions. This is accomplished by specifying PUSH/POP addressing in the appropriate operand field. If this addressing is specified within a source operand, the processor will execute a POP operation in which the contents of the data field located on the top of the process stack will be transferred into an internal data register. The TOS register will then be incremented by an amount determined by the length of the data field. The TOS Register then points to the next data field on the stack. If PUSH/POP addressing is specified in a destination operand, the results of the instruction execution will be transferred onto the process stack. As with all process stack operations, the TOS

register is adjusted to point to the last byte of data which was placed onto the stack. When PUSH/POP addressing is selected, the Auxiliary Data Field specifies the Data Access Mode in exactly the same manner as that described above for the primary addressing modes.

The auxiliary data field (bits 1 through 0) of the operand control byte are defined as follows for PUSH/POP operations:

bit 1	bit 0	Operation
0	0	PUSH/POP a variable defined by a descriptor (PPD).
0	1	PUSH/POP byte (1 byte) (PPB).
1	0	PUSH/POP half word (2 bytes) (PPHW).
1	1	PUSH/POP triple byte (3 bytes) (PPTB).

The PUSH/POP operations are illustrated in Figure 2.1 with the following instruction:

```
    .
    .
    .
    .
Add     PPD,PPB,PPHW    ;POP descriptor variable and byte
  .                     ;variable from stack, add them
  .                     ;together and PUSH the results onto
  .                     ;the stack in a two byte
  .                     ;integer field.
  .
  .
```

Figure 2.1a shows the process stack before the add instruction. Note the top two elements of the stack are a data field with descriptor (Data Field # 1) and a single byte of data (Data Field # 2). The TOS register initially points to the descriptor of the Data Field # 1.

Figure 2.1b shows the process stack after the PPD POP operation. After the operation is complete, the contents of this data have been transferred into an internal data register and the TOS register has been adjucted to point to Data Field #2. Note that the POP operation does not change the contents of the memory locations in which Data Field #1 is stored.

After the PPB (POP) operation, illustrated in figure 2.1C, the TOS has been incremented by 1 and now points to earlier data stored on the stack.

The result of the add operation will be pushed onto the stack as a

two byte integer. Note the TOS register has been adjusted to point to the last byte of data accessed on the stack. Figure 2.1d depicts the PPHW (PUSH) operation.

Figure 2.1A - Memory and TOS register contents before POP

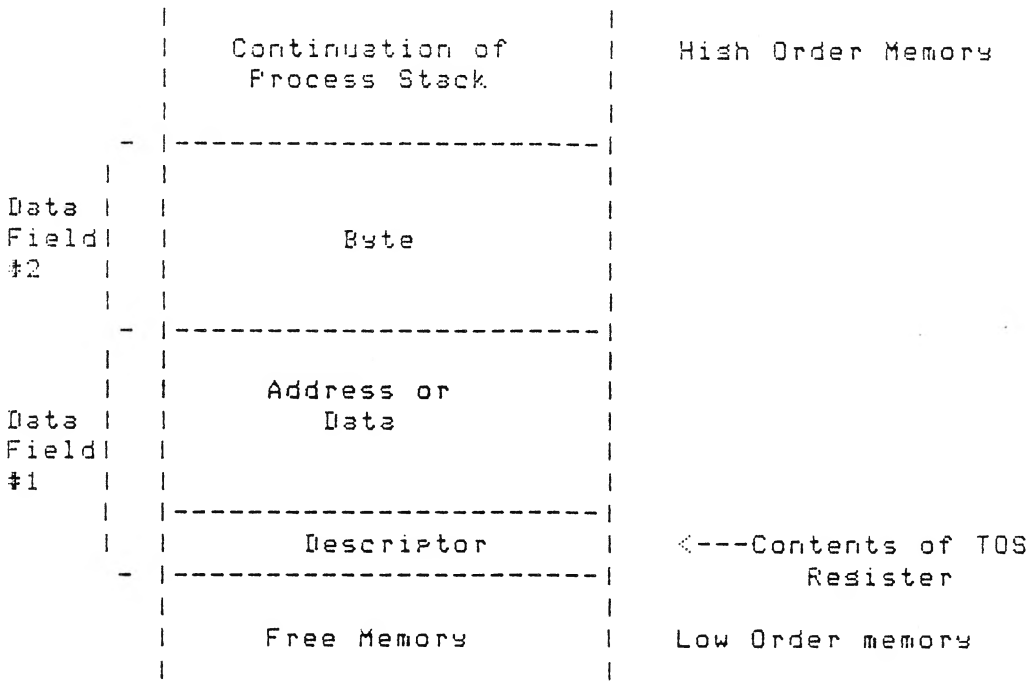


Figure 2.2B - Memory and TOS register contents after PPD (POP)

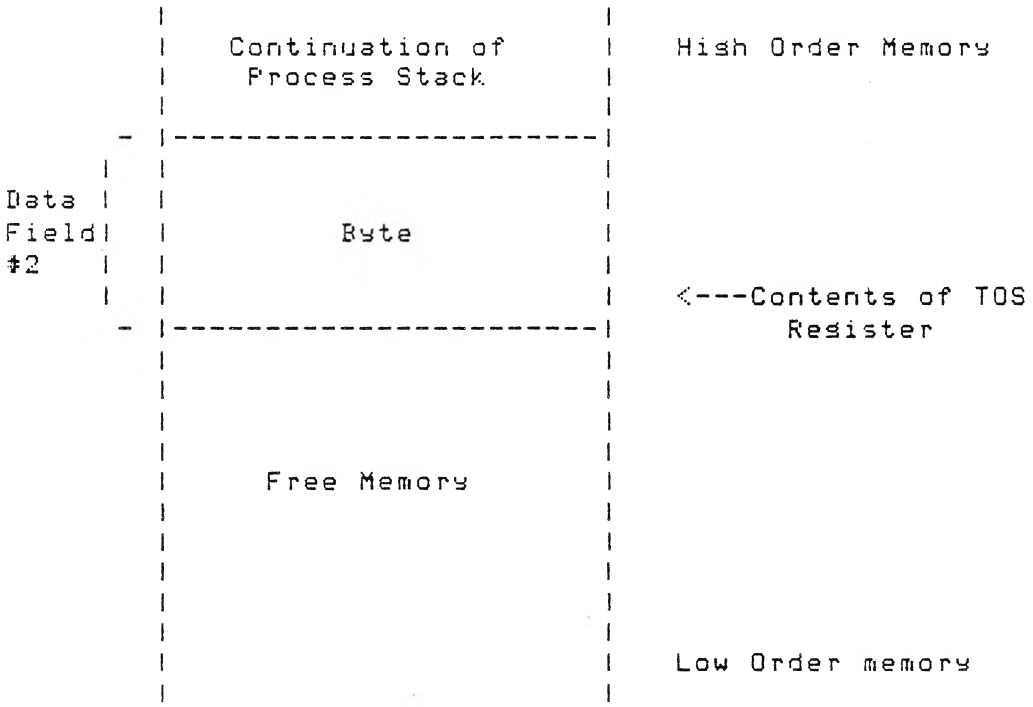


Figure 2.1C - Memory and TOS register contents before PPB (POP)

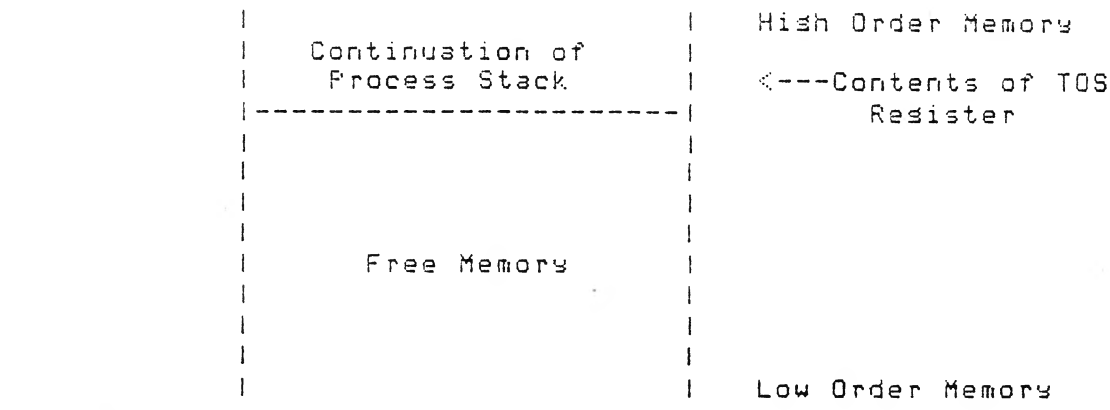
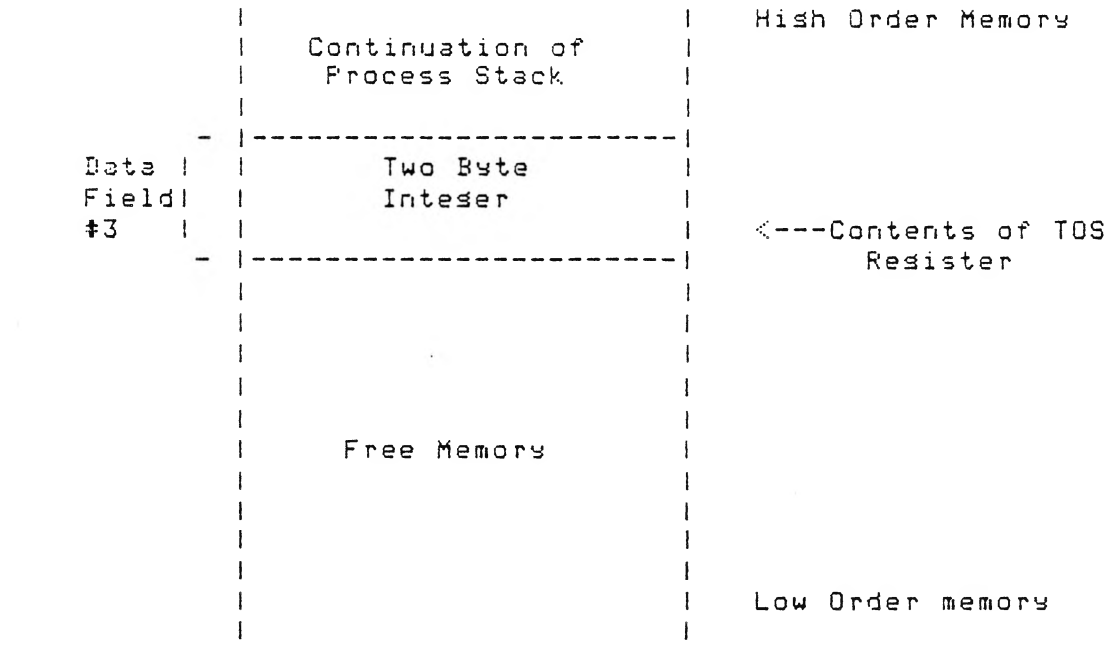


Figure 2.1D - Memory and TOS register contents after PPHW (PUSH)



4.5.3.4 Immediate Addressing, Long Form

The long form of immediate addressing allows up to three bytes of data or addressing information to be specified within the instruction. This data follows directly behind the Addressing Control Byte. When this form of addressing is selected, the Auxiliary Data Field (bits 0 and 1) specifies the number of bytes and the format of the immediate data as follows:

Bit 1	Bit 0	Format of Data
0	0	Unsigned Byte. Assumed to be positive.
0	1	Signed 2 byte integer.
1	0	Three byte ordinal. Assumed to be positive.
1	1	Not Used

4.5.4 Internal Register Addressing

This addressing mode allows the internal processor registers can be specified as the source or destination of the data to be manipulated by the instruction. The four-bit Register Select data field selects the internal registers as follows:

Bit 3	Bit 2	Bit 1	Bit 0	Register Selected
0	0	0	0	Process Limit Register
0	0	0	1	Process Program Counter
0	0	1	0	Top Of Stack Register
0	0	1	1	Primary Base Register
0	1	0	0	Process Control register
0	1	0	1	Microcode Select Register
0	1	1	0	Refresh Control Register

The LMT register is 'read-only', i. e., the process software cannot modify the contents of this registers under any conditions. The Process Control Register, Microcode Select Register, and Refresh Control Register are 'read-only' in the User Mode and 'Read / Write' in the Supervisor Mode.

4.5.5 Immediate Addressing, Short Form

The first of the short form addressing modes allows five bits of immediate data to be included in the Operand Control Byte (in bits

0 - 4). This allows immediate values between -16 and +15 to be specified within the Operand control Byte.

4.5.6 Process Base Addressing, Short Form

The short form of BAS offset addressing allows up to 6 bits of offset information to be included in the Operand Control Byte (in bits 0 - 5). This allows the Operand Control Byte to directly specify data fields which are accessed through descriptors located in the first 64 bytes of the Base Page.

4.5.6 Primary Base Addressing Short Form

The short form of PRM offset addressing allows up to 6 bits of addressing information to be included in the Addressing Control Byte (in bits 0 - 5). This allows the Operand Control Byte to directly control access to data fields which are accessed through descriptors located in the first 64 bytes above that memory location whose address is contained in the Primary Base Register.

Figure 4.6A and 4.6B below summarizes the addressing modes for the MCS65E4.

< > : Specify the valid data types

( ) : Terminal symbols which specify the form of the data field.

[ ] : Reference a more detailed syntax diagram by that name.

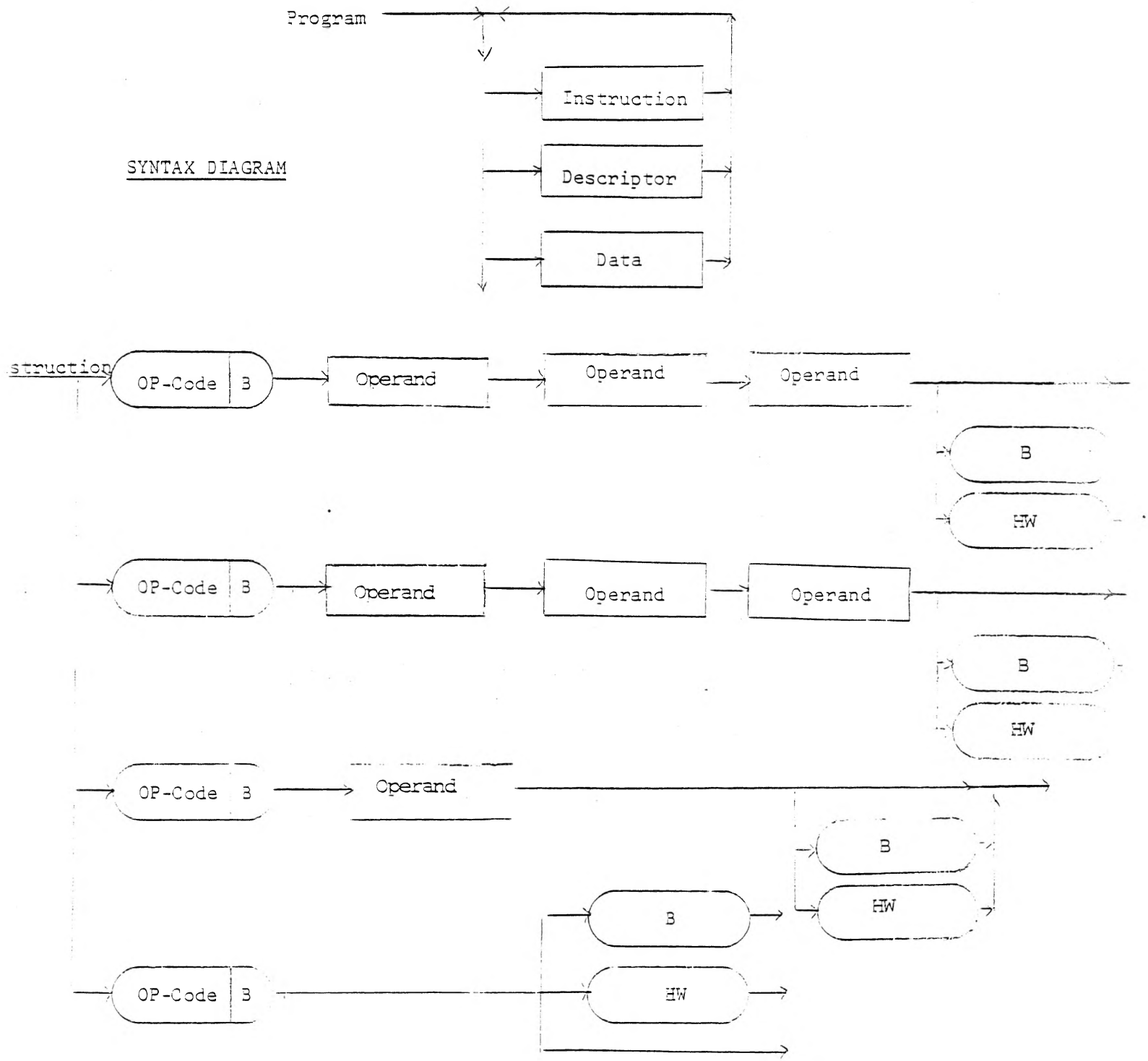


Figure 5.6a

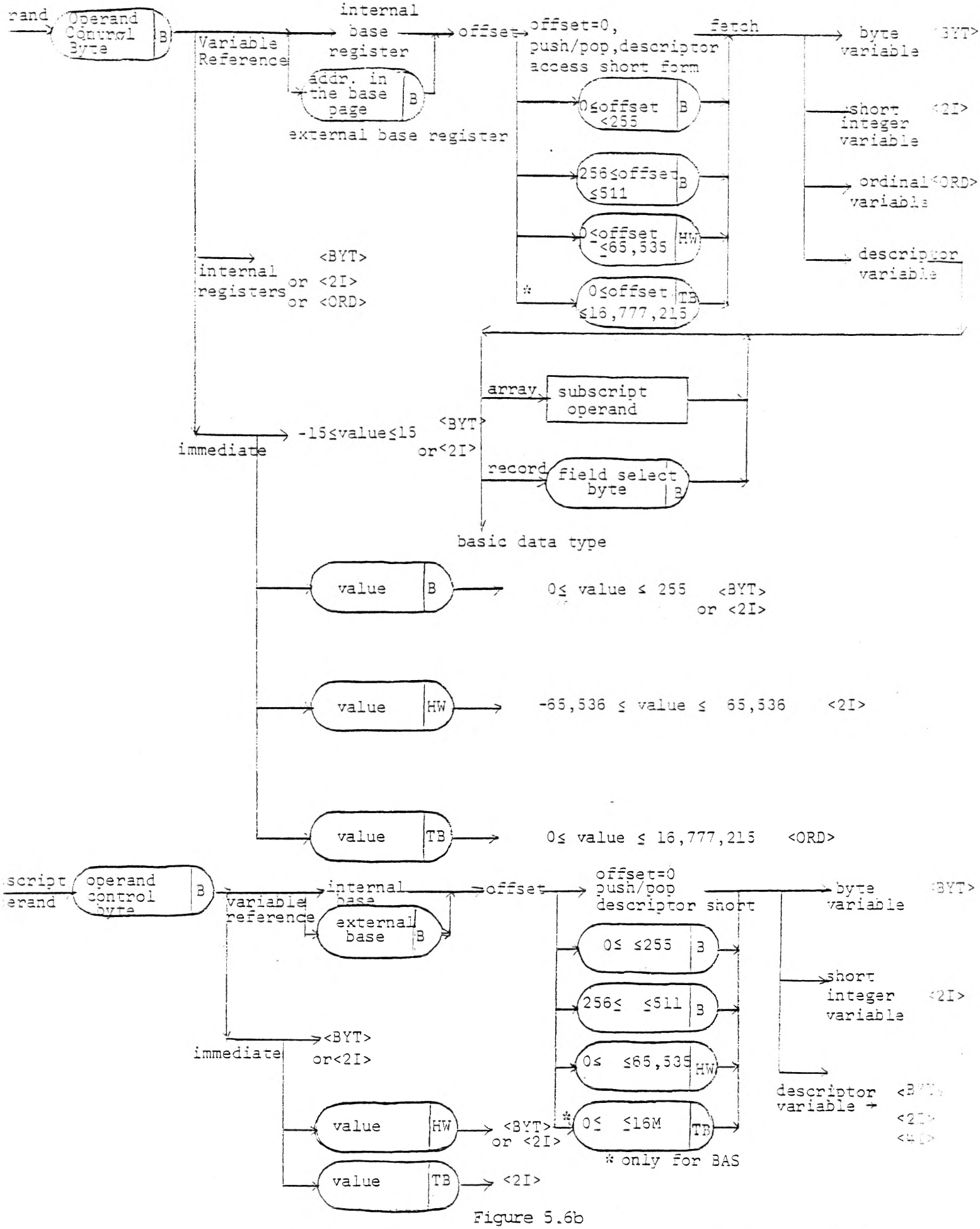

address calculation
= (BASE) + offset
rand
Operand Control Byte
B
Variable Reference
internal base register
addr. in the base page
B
external base register
offset
offset=0, push/pop,descriptor access short form
fetch
byte variable
<BYT>
short integer variable
<2I>
ordinal
<ORD>
variable
descriptor variable
0≤offset <255
B
256≤offset ≤511
B
0≤offset ≤65,535
HW
*
0≤offset ≤16,777,215
TB
internal registers
<BYT>
or <2I>
or <ORD>
array
subscript operand
immediate
-15≤value≤15
<BYT>
or<2I>
record
field select byte
B
basic data type
value
B
0≤ value ≤ 255
<BYT>
or <2I>
value
HW
-65,536 ≤ value ≤ 65,536
<2I>
value
TB
0≤ value ≤ 16,777,215
<ORD>
script
erand
operand control byte
B
variable reference
internal base
external base
B
offset
offset=0
push/pop
descriptor short
byte variable
<BYT>
short integer variable
<2I>
descriptor variable
<BYT>
<2I>
<4I>
0≤ ≤255
B
256≤ ≤511
B
0≤ ≤65,535
HW
*
0≤ ≤16M
TB
* only for BAS
immediate
<BYT>
or<2I>
value
HW
<BYT>
or <2I>
value
TB
<2I>

Figure 5.6b

4.6 Data Structure Within the MCS65E4 System

4.6.1 Introduction

Nowhere is the 'high level' approach to architecture more apparent than in the manner in which data is stored and accessed within an MCS65E4 system. This was introduced in Section 4.5 which discusses the organization of the operand. Section 4.6 contains a detailed description of the remaining aspects of these data accessing mechanisms along with a description of the manner in which data is stored within the MCS65E4 system.

The principal feature of the MCS65E4 data structure is the use of descriptors to specify all of the pertinent details concerning a data field. This contrasts sharply with the conventional approach in which the field length is determined primarily by the Op Code (8 bits, 16 bits, etc.) and the actual data fields are created by utilizing the processor software to organize these simple fixed length fields into groups to store complex data entities (10-byte Real, 8-byte BCD, etc.).

The principal advantage associated with this traditional approach is that it is much simpler to implement in hardware. The instruction set consists of a large number of relatively simple operations. This is compatible with an environment in which hardware is expensive, logic design is still somewhat unsophisticated, and the science of system programming is still in its infancy. Implementation of this type of architecture at this time, however, does not recognize the significant developments which have taken place in processor design techniques (multi-level microprogramming, etc.). In addition, it ignores the fact that software design techniques have achieved a substantial degree of maturity. Specifically, the compiler languages in use today exhibit the following characteristics :

1. The algorithms and data structures are kept as separate, self-contained entities. For example, the statement A=B+C typically contains no information regarding the type of data which is stored in A, B or C. Instead, the data type is defined earlier in the program as integer, real, etc.

2. Data elements are treated as complete entities most of the time. This means, for example, that the various segments of a floating point variable will not be treated individually by a user's application program.

3. In most instances, the form of the data within a data field will not change during the life of a program.

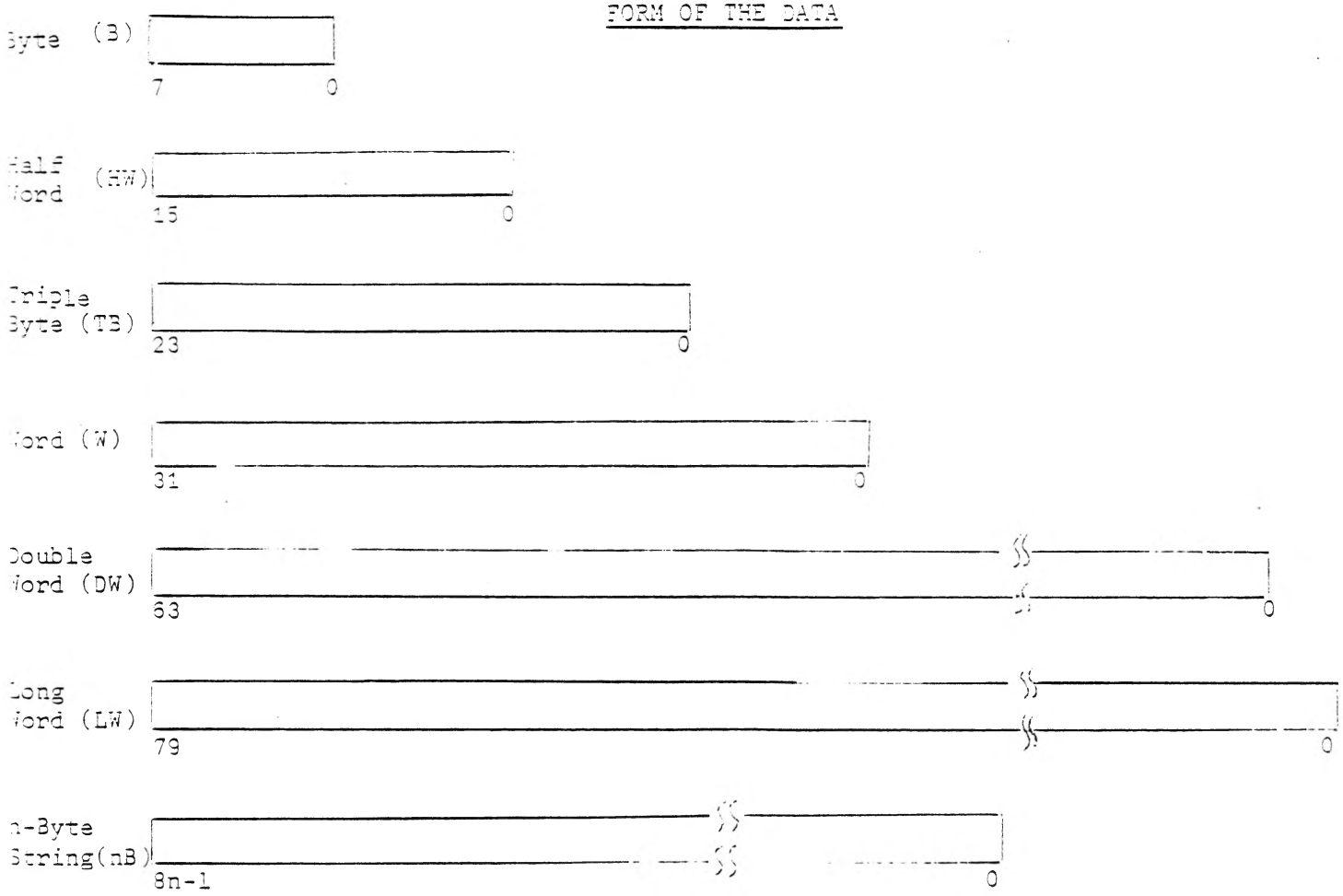
4. The operations which are performed on a data field are generally a function of the specific data contained in the field, for example, the arithmetic operations which are performed on a floating point data field will differ sharply from those which are performed on an integer data field.

The architecture of the MCS65E4 is designed in a manner which recognizes these characteristics. However, it also recognizes the use of the phrases 'most of the time', 'in most instances', and 'generally' in the above list. Specifically, it provides all of the advantages of descriptors while still retaining the flexibility needed by such languages as Fortran, and C in which the definition of a data field can be altered during the execution of a program. In addition, it recognizes the need to allow efficient manipulation of the descriptor and other key elements of the data structure.

All data manipulation instructions within the MCS65E4 consist of an Op Code and up to three operands. The Op Codes specify the operation to be performed while the operands specify the location of the data to be manipulated. This data can be accessed directly or it can be located in one of the complex data structures described below. In all cases, however, processing of the operand must result in the generation of the address of a basic data element since all data manipulation operations are performed only on these elements.

Within the MCS65E4 system most data is accessed through descriptors. Hereafter, these descriptors contain the type, format and location information which allows the processor to manipulate the data in a data field. After a brief description of the basic data elements, the organization and operation of these variable descriptors are discussed in detail.

FORM OF THE DATA



	Data Type	Meaning	Size	Range
1	BYT	byte	B	Binary = 0 to 255 BCD = 0 to 99
2	2I	2 byte signed integer	HW	-65,536 to 65,535
3	ORD	Ordinal-3 byte unsigned integer	TB	0 to 16,777,215
4	4I	4 byte signed integer	W	-2^{31} to $2^{31}-1$
5	4R	4 byte real	W	$\pm 10^{-37}$ to 10^{+38} 7 decimal digit of precision
6	4D	4 byte BCD	W	$\pm$7 decimal digit
7	8I	8 byte integer signed	DW	-2^{63} to $2^{63}-1$
8	8R	8 byte real	DW	$\pm 10^{-308}$ to 10^{308} 15 decimal digit of precision
9	8D	8 byte BCD	DW	$\pm$15 decimal digit
10	10R	10 byte real	LW	$\pm 10^{-4932}$ to 10^{4932} 19 decimal digit of precision
12	STR	String	0 to 32767 bytes	Binary: 0 to $2^{262,136}$ Decimal: 0 to $10^{65,534}$

Note: 1-11 will be refered to as scalar

4.6.2 The Basic Data Elements

The group of basic data elements is composed of 10 simple data fields and 1 string data field. The simple data fields (hereafter referred to as scalers), consist of signed and unsigned binary data, BCD data, or floating point (REAL) data in varying length fields. Each of these Basic Data Elements is stored in one of seven different field sizes (Byte, Half Word, Triple Byte, Word, Double Word, Long Word, n-Byte String) which is depicted in Figure 5.7.2a

4.6.2.1 Unsigned Binary Data Fields

The unsigned binary data fields are type byte and ordinal. They are assumed to be positive and thus have no sign information associated with them. A byte is 8 bits and an ordinal is 24 bits. The range of these field sizes are indicated in figure 5.7.2b.

4.6.2.2 Signed Binary Data Fields

The signed binary data fields are two, four, and eight byte integer. They are stored in two's compliment binary format. The range of these fields are indicated in figure 5.7.2B.

4.6.2.3 Binary Coded Decimal (BCD) Data Fields

The MCS65E4 supports four and eight byte BCD fields stored as packed binary integer and signed magnitude. The most significant four bits of the field contains the sign information (i.e 0000 => positive number, 1111 => negative number). The range of these fields are indicated in figure 5.7.2b.

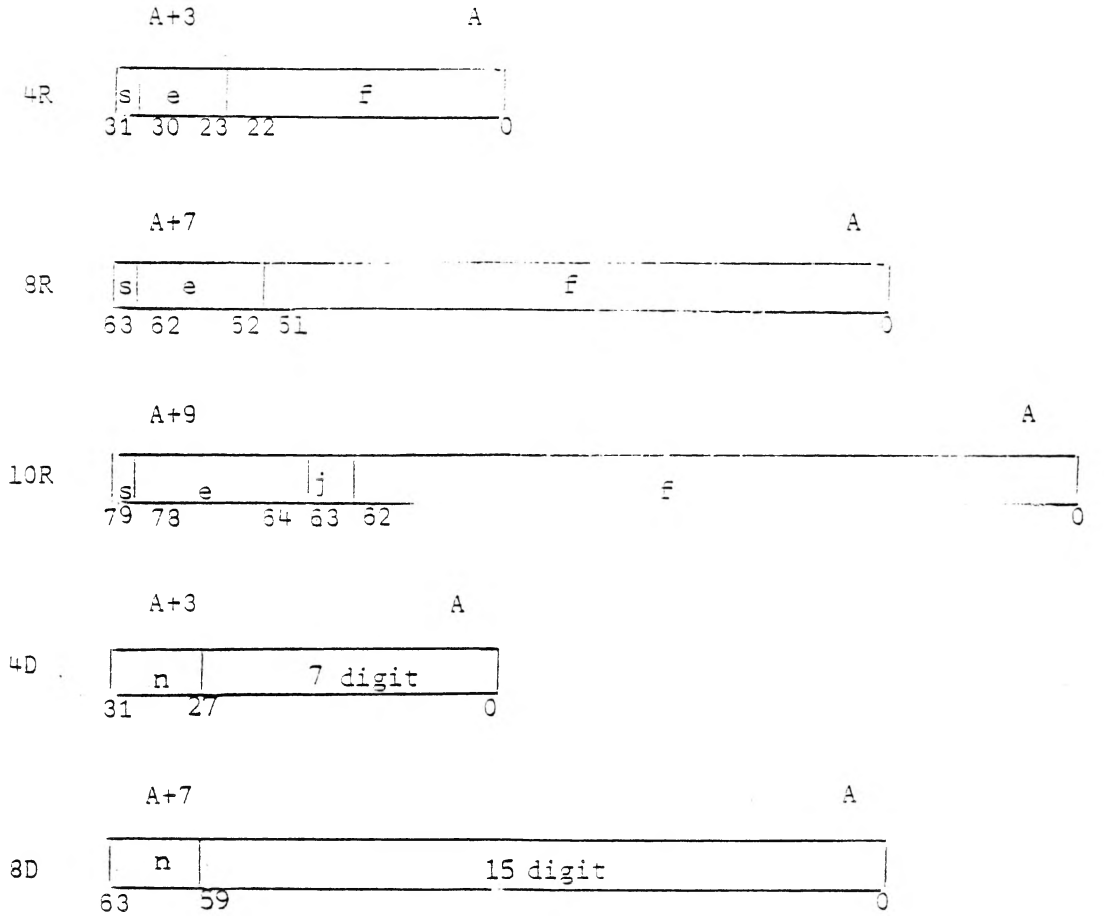
4.6.2.4 Floating Point Data Fields

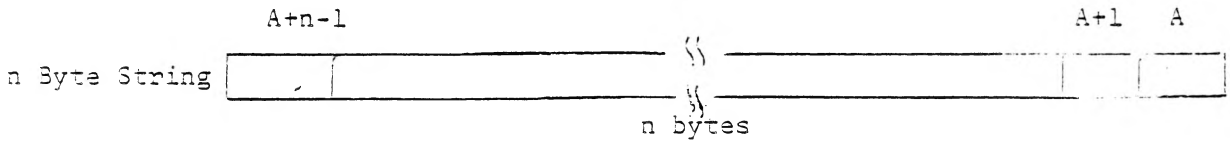
The floating point data fields are compatible with the proposed IEEE floating point standard. The are stored in three parts: mantissa, biased exponent, and sign. Both the mantissa and the exponent are stored in two's compliment binary format with the most significant bit being the sign indicator (i.e. 0 => positive, 1 => negative). The range of these fields are indicated in figure 5.7.2B.

4.6.2.5 String Data Fields

String data fields are treated as unsigned binary data.

Figure 5.7.2C depicts the format of the Basic Data Elements





s = sign

e = exponent

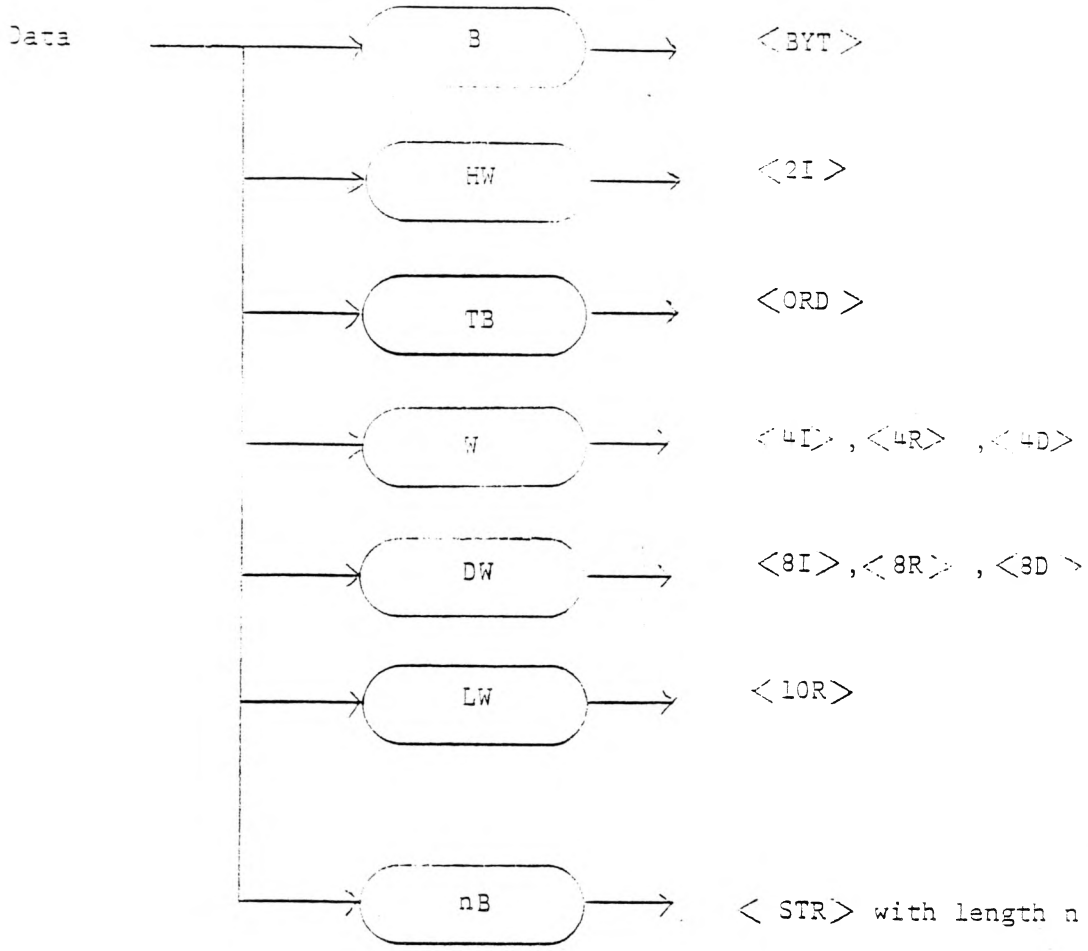
f = fraction

K = key

M = message

j = 1-bit integer part

n = sign 0000
1111


Data
B
<BYT>
HW
<2I>
TB
<ORD>
W
<4I>, <4R>, <4D>
DW
<8I>, <8R>, <8D>
LW
<10R>
nB
<STR> with length n

4.6.3 Organization of the Variable Descriptor

4.6.3.1 Introduction

The Variable Descriptor is the primary means by which the MCS65E4 determines the format of the data field. With the exception of the one byte unsigned binary field, the two byte signed integer field, and the three byte ordinal which can be accessed directly (without going through a descriptor), all data fields must be accessed through a descriptor. This assures that all instructions will be executed in a manner which is appropriate to the type of data which is being manipulated.

The Variable Descriptor is composed of one or more of the following elements:

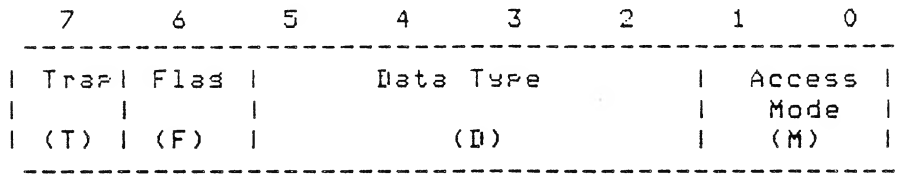
1. Descriptor Header
2. Addressing Information (optional)
3. Auxiliary information (optional)

All Variable Descriptors must begin with a Descriptor Header. However, the addressing information must be provided only when the data field is not attached directly to the descriptor and the auxiliary data must be provided only when referencing the more complex data structures (strings, records, etc.).

4.6.3.2 Organization of the Descriptor Header

4.6.3.2.1 Introduction

The organization of the Descriptor Header can be depicted as follows:



Each of these fields is described briefly below. A detailed description of the operation of the descriptor is contained in the remaining paragraphs of this section.

4.6.3.2.2 Trap Bit (Bit 7)

The trap bit in the descriptor can be set to cause a Data Access Trap to occur when the processor attempts to access the data field. Within the Exception Vector field, the TRAP bit is used to specify that the exception will be serviced in the current process's caller.

4.6.3.2.3 Access Mode (Bits 1 and 0)

The Access Mode field specifies the manner in which the location

of the data field will be determined. Specifically, the data can be attached to the descriptor, it can be located at a specified offset from the descriptor or it can be located at a specified logical address within the process. The two bits in this field specify one of four access modes as follows:

Bit 1	Bit 0	Descriptor	Field Descriptor or Element Descriptor
0	0	Attached	Attached Relocatable
0	1	Short Relative	Short Relocatable
1	0	Long Relative	Long Relocatable
1	1	Logical Addressing	Logical Addressing

4.6.3.2.3.1 Attached

If the Attached Access Mode is specified, the variable descriptor consists of a descriptor header and, if appropiate, an auxiliary data field. No addressing information is required since the data field is located in memory immediately following the descriptor.

4.6.3.2.3.2 Attached Relocatable

Within an element descriptor in an array structure or a field descriptor in a record, the attached addressing becomes attached relocatable. The attached relocatable addressing mode operates in much the same manner as attached except that the data is assumed to be attached to the address which was previously calculated during the descriptor processing. This is illustrated in detail in the examples below.

4.6.3.2.3.3 Short Relative

If the short relative access mode is specified, the data field is located at a specified offset from the address of the descriptor. The offset is contained in a single byte of addressing information immediately following the descriptor header. The data field can therefore be located within the range of -128 to +127 bytes from the address of the descriptor. If auxiliary information is required for accessing the data field, this information follows immediately after the single byte of offset. (refer to descriptor flowchart)

4.6.3.2.3.4 Short Relocatable

The short relocatable access mode is exactly the same as short relative addressing mode except that the offset information is added to the 'previously calculated address' within an array structure or record. See example below for accessing data within a multi-dimensional array or record.

4.6.3.2.3.5 Long Relative

If the long relative access mode is specified, two bytes of offset infomation are included in the variable descriptor. The descriptor header is followed by the low order and high order bytes of a

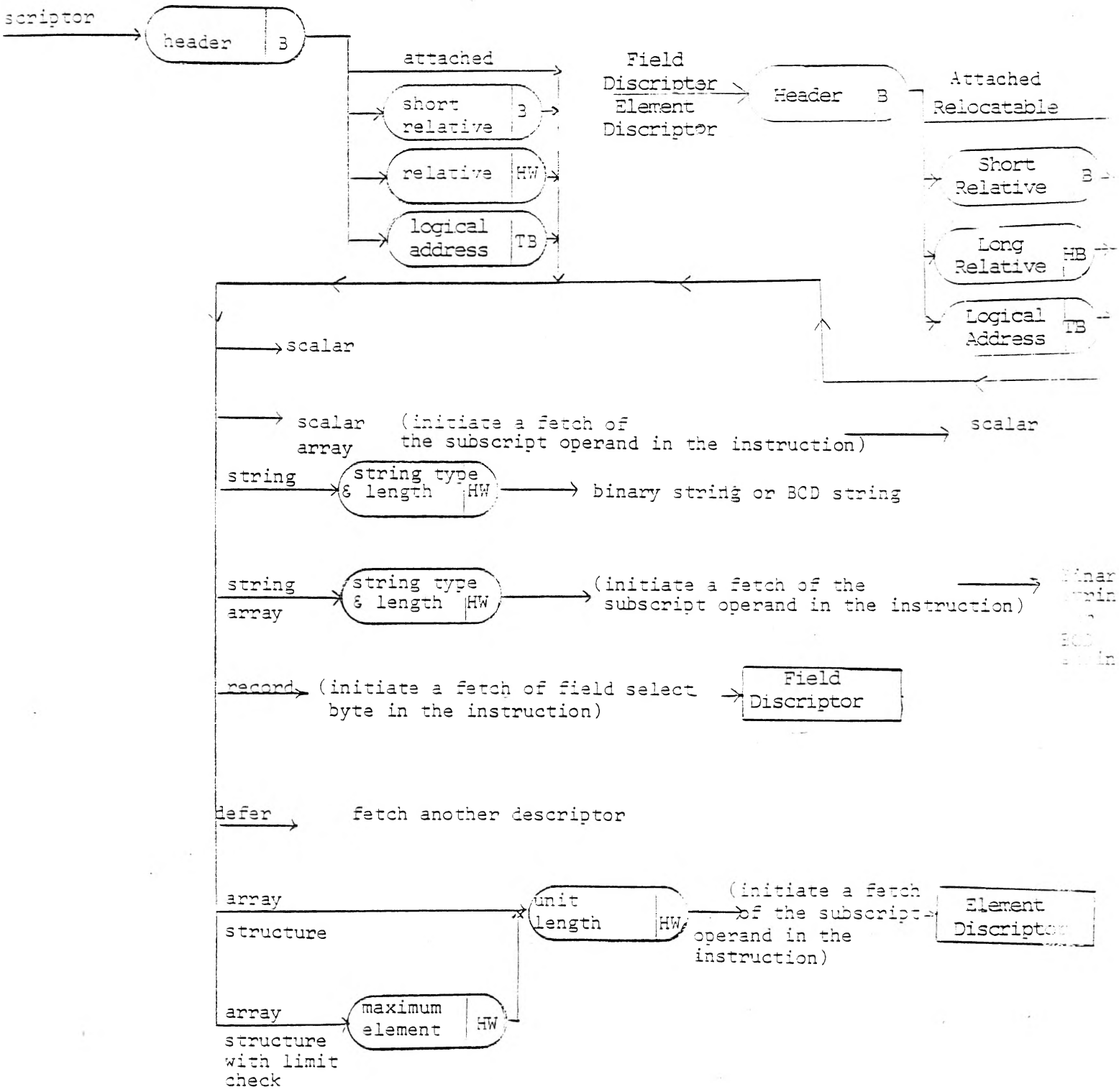
16-bit offset field. This allows the data field to be located within a range of -32868 to +32767 bytes from the address of the descriptor. If the data field is one of the more complex structures described below, the required auxiliary data will follow the two byte offset in the descriptor.

4.6.3.2.3.6 Long Relocatable

The long relocatable access mode is exactly the same as long relative addressing mode except that the offset information is added to the 'previously calculated address' within an array structure or record. See the example below for accessing data within a multi-dimensional array or record.

4.6.3.2.3.7 Logical Addressing

If Logical Addressing is specified, the descriptor header will be followed by a three-byte ordinal which is the logical address of the data field. This logical address is added to the contents of the BAS register to determine the physical address of the data field. This allows the data field to be placed anywhere in the address space of the process. As before, any auxiliary information which is required will follow the three bytes of addressing information. (Refer to descriptor flowchart)


scriptor
header B
attached
short relative B
relative HW
logical address TB
Field Discriptor Element Discriptor
Header B
Attached Relocatable
Short Relative B
Long Relative HB
Logical Address TB
scalar
scalar array
(initiate a fetch of the subscript operand in the instruction)
scalar
string
string type & length HW
binary string or BCD string
string array
string type & length HW
(initiate a fetch of the subscript operand in the instruction)
record
(initiate a fetch of field select byte in the instruction)
Field Discriptor
defer
fetch another descriptor
array
structure
unit length HW
(initiate a fetch of the subscript operand in the instruction)
Element Discriptor
array
structure with limit check
maximum element HW

4.6.3.2.4 Data Type Field (bits 5 - 2) and Flag (bit 6)

The Data Type Field and the Flag operate together to specify the exact nature of the data field. The D field specifies the size of the field and the type of data stored in the field as follows:

**** Bit **** 5	4	3	2	Data Type
0	0	0	0	Not Used (Reserved for future expansion)
0	0	0	1	Byte (BYT)
0	0	1	0	Ordinal (ORD)
0	0	1	1	Two-Byte Integer (2I)
0	1	0	0	Four-Byte Integer (4I)
0	1	0	1	Eight-Byte Integer (8I)
0	1	1	0	Four-Byte Real (4R)
0	1	1	1	Eight-Byte Real (8R)
1	0	0	0	Ten Byte Real (10R)
1	0	0	1	Four-Byte BCD (4D)
1	0	1	0	Eight-Byte BCD (8D)
1	0	1	1	String (STR)
1	1	0	0	Deferred Descriptor/Record
1	1	0	1	Array Structure
1	1	1	0	Not Used (Reserved for future Expansion)
1	1	1	1	Not Used (Reserved for future Expansion)

BYTE SIZE D A T A E L E M E N T S

For the first 11 items in this table (byte through string), setting the FLAG to a logic 1 specifies that the data field is an element in a single dimension array. In this case, the processor will fetch the next operand from the instuction and use the value as the index information to determine the location of the data in the array. If the FLAG is a logic 0, the descriptor points directly to the data field.

If the Data Type field specifies that the data field is an array structure (D=1101), the FLAG is used to enable and disable automatic checking of the index contained in the instruction to assure that it is not less than 0 or greater than the maximum value specified in the descriptor. If the Data Type field contains a binary 1100 (Deferred Descriptor/record), the FLAG is used to select between the Deferred Descriptor and the Record.

The following table summarizes the use of the FLAG bit within the descriptor header:

Data Type	FLAG Value	Definition
Byte	0	Descriptor Access

	through Strins	1	Array Access
Array Structure		0	No Range Checking
		1	Perform Range Checking
Deferred Descriptor or record		0	Record
		1	Deferred Descriptor

The data type classifications depicited above can be organized into three groups. The first group contains all of the basic data elements which are the data fields that are operated upon by all of the data manipulation instructions. The elements in this group are identified by a 0001 through 1011 in the Data Type field (byte through strins). The second group contains all of the simple data structures (arrays, records, etc.), while the third contains the Deferred Descriptor.

The sequence of operations which the MCS65E4 goes through to access a basic data element is shown below (refer to the flow diagram - the sequence numbers corresponds to the diagram).

1. The processor accesses the descriptor block through any of the MCS65E4 addressins mode (see section _____). Set Y = the address followins the addressins information.

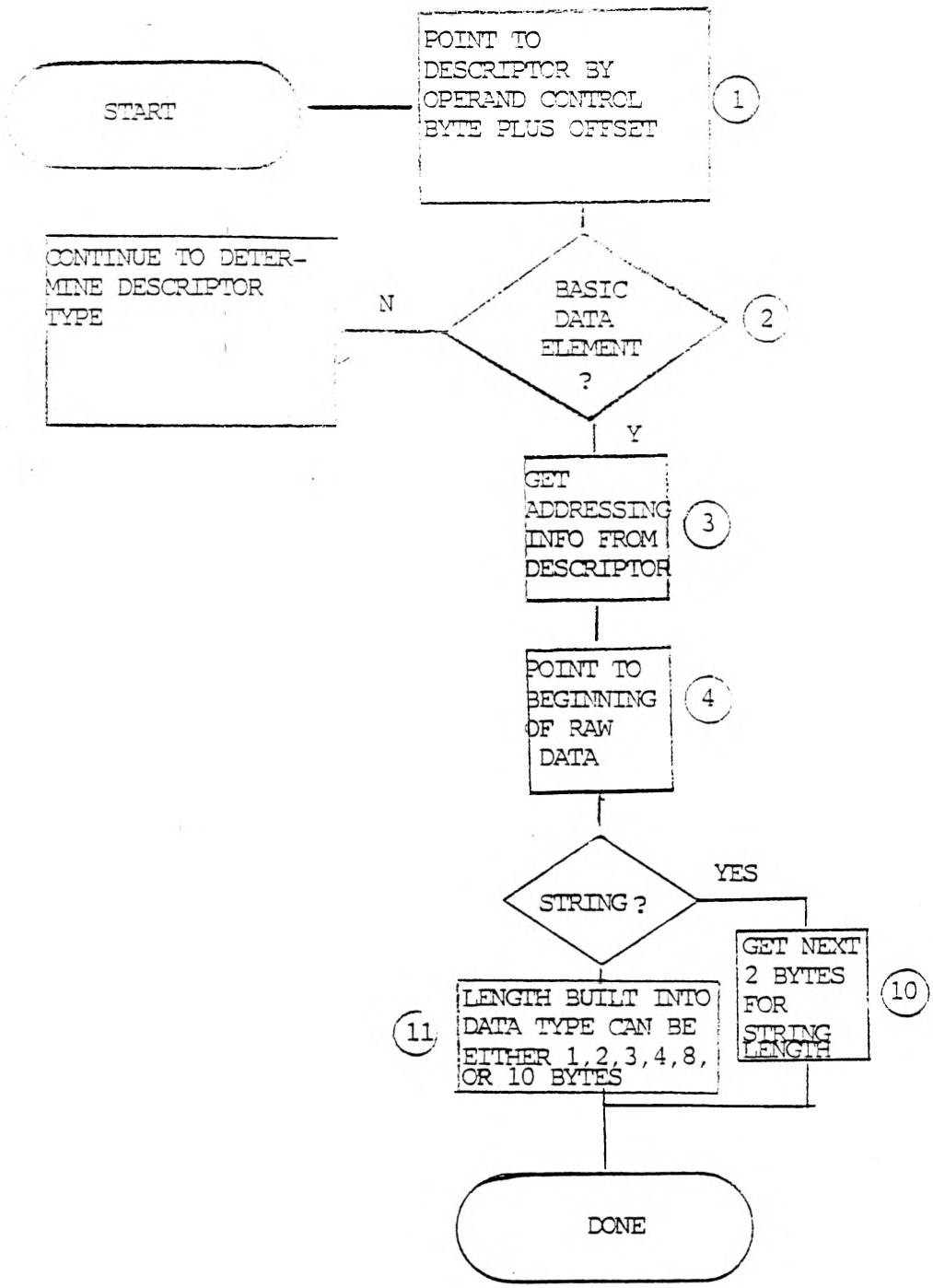
2. A check is made to determine whether the descriptor indicates an basic data type. If it is not, the processor continues to determine the desecriptor's type.

3. If the descriptor indicates basic data type, the MCS65E4 sets the addressins information from the descriptor block.

4. The addressins information and the access mode bits together determine the startins address of the raw data. (In the flow diagram, this calculated address is saved in variable Y - an internal resister. For attached mode, Y = Y + 0. For relocatable mode, Y = Y + relative. For losical mode, Y = the physical address.)

10. A check is made to determine if the basic data is of type strins. If it is, the next two bytes are fetched from the instruction and are used as the strins lensth.

11. If the basic data type is not type strins the lensth is determined by the data type. The lensth can be either 1, 2, 3, 4, 8, or 10 bytes.


START
POINT TO DESCRIPTOR BY OPERAND CONTROL BYTE PLUS OFFSET
1
CONTINUE TO DETERMINE DESCRIPTOR TYPE
N
BASIC DATA ELEMENT ?
2
Y
GET ADDRESSING INFO FROM DESCRIPTOR
3
POINT TO BEGINNING OF RAW DATA
4
STRING ?
YES
GET NEXT 2 BYTES FOR STRING LENGTH
10
11
LENGTH BUILT INTO DATA TYPE CAN BE EITHER 1,2,3,4,8, OR 10 BYTES
DONE

4.6.4 The Data Structures

4.6.4.1 Introduction

In addition to the simple data fields described previously, the MCS65E4 directly supports the storage of data in a number of simple data structures. These are:

1. Single-dimension arrays
2. Array Structures
3. Records

In addition, these structures can be organized into complex data structures such as multi-dimensional arrays, arrays of records, etc. The simple data structures are described in this section. The manner in which these simple data structures can be used to build the complex data structures is best illustrated by example.

4.6.4.2 Single-Dimension Arrays

The single-dimension array is the simplest of the data structures which are directly supported by the MCS65E4. This is selected by setting the FLAG bit in the variable descriptor to a logic 1. The processor then assumes that the scalar or string which is to be manipulated by the instruction is an element of an array. To access this element, the processor fetches an index from the instruction.

The sequence of operations which the MCS65E4 goes through to access an element within a single dimension array is as follows: (refer to the flow diagram - the sequence numbers corresponds to the diagram).

1. The processor accesses the descriptor block through any of the MCS65E4 addressing mode. Set Y = the address following the addressing information.

2. A check is made to determine whether the descriptor indicates single dimesion array. If it is not, the processor continues to determine the desecriptor's type.

3. If the descriptor indicates single dimension array (FLAG=1), the MCS65E4 sets the addressing information from the descriptor block.

4. The addressing information and the access mode bits together determine the starting address of the raw data for the first element of the array (In the flow diagram, this calculated address is saved in variable Y - an internal register. For attached mode, Y = Y + 0. For relocatable mode, Y = Y + relative. For logical mode, Y = the physical address.)

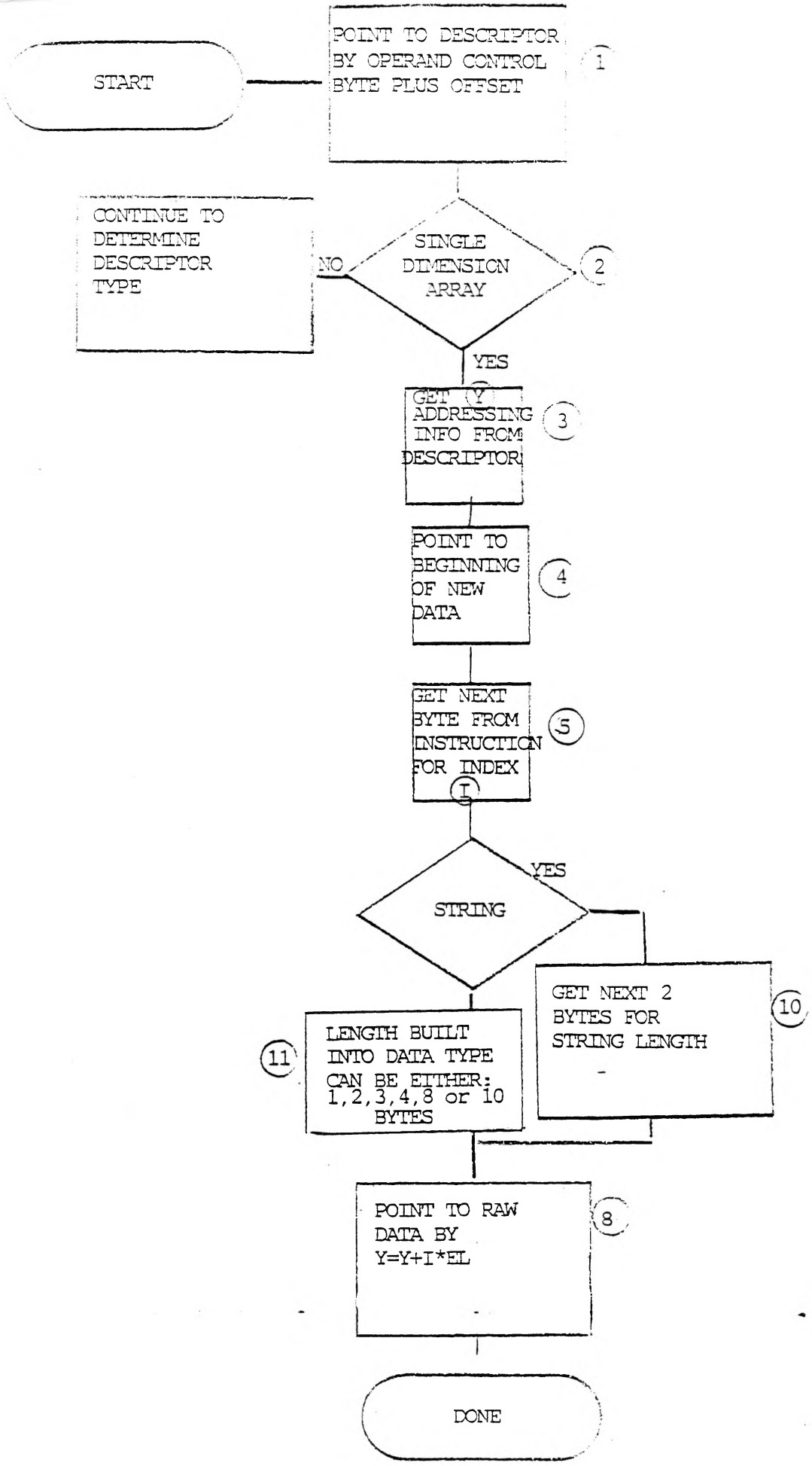
5. To determine which element in the array to access, the processor fetches the data specified by the next

operand in the instruction. This data is treated as the index (In the flow diagram this value is referred as variable I).

10. A check is made to determine if the basic data is of type string. If it is, the next two bytes are fetched from the instruction and are used as the string length.

11. If the basic data type is not type string the length is determined by the data type. The length can be either 1, 2, 3, 4, 8, or 10 bytes.

8. The element length (EL), the index value (I), and the previous calculated address (Y) are used to address the sought after element within the array structure. The formula for the address calculation is Y = Y + (I * EL). Note that the location of this element can be determined without regard for the exact nature of the data being stored in the array.

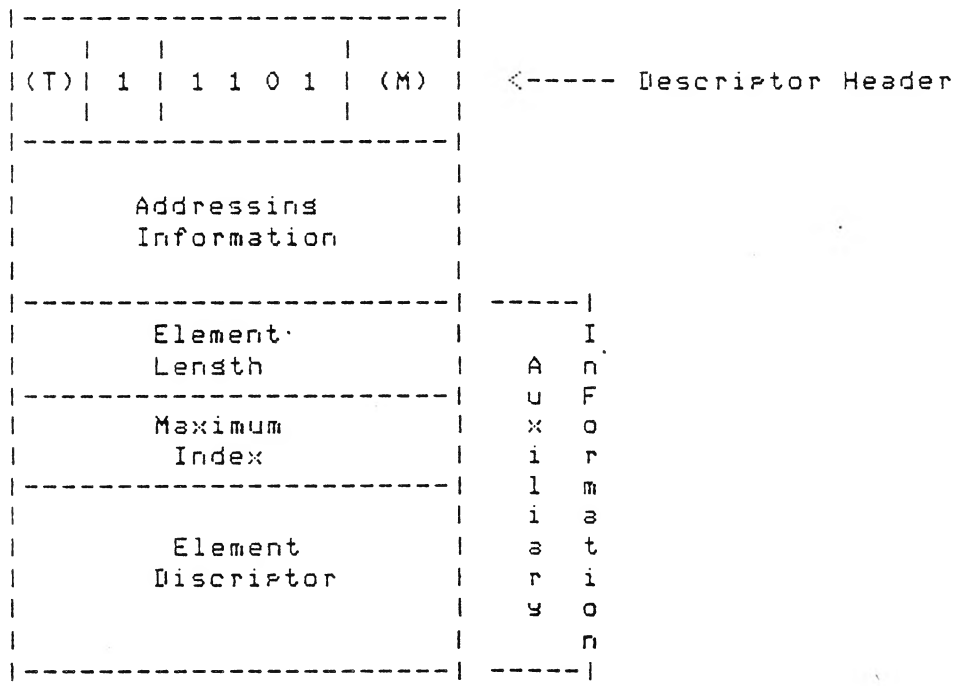

START
POINT TO DESCRIPTOR BY OPERAND CONTROL BYTE PLUS OFFSET
1
CONTINUE TO DETERMINE DESCRIPTOR TYPE
NO
SINGLE DIMENSION ARRAY
2
YES
GET (Y) ADDRESSING INFO FROM DESCRIPTOR
3
POINT TO BEGINNING OF NEW DATA
4
GET NEXT BYTE FROM INSTRUCTION FOR INDEX (I)
5
YES
STRING
GET NEXT 2 BYTES FOR STRING LENGTH
10
11
LENGTH BUILT INTO DATA TYPE CAN BE EITHER: 1,2,3,4,8 or 10 BYTES
POINT TO RAW DATA BY Y=Y+I*EL
8
DONE

4.6.4.3 Array Structure

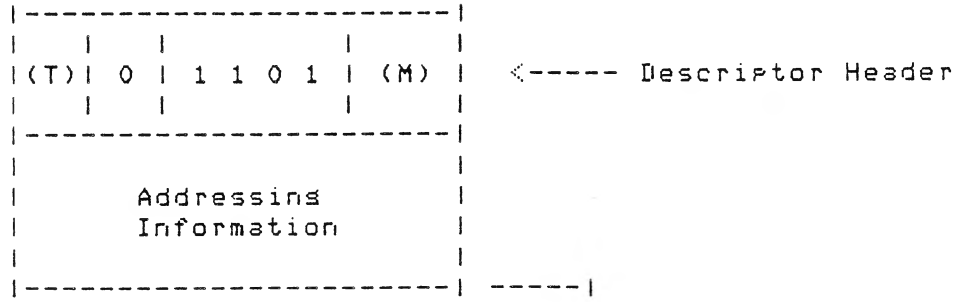
In addition to the simple array described in the previous paragraph, the MCS65E4 supports the storage of data in arrays in which elements can be more than the basic data elements. This is termed the 'array structure' and is selected by setting the Data Type to 1101 (binary).

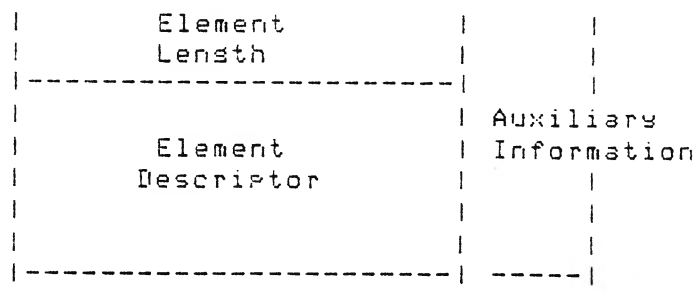
When an array structure is specified, the descriptor header contains no information regarding the type of data which is stored in the array. Instead, this is contained in the auxiliary data field which follows the addressing information. Also, when FLAG equals 0 (i.e. no limit checking), the maximum index field within the array descriptor block is omitted.

The format for the descriptor block, with and without limit checking, which references an array structure is shown below:



Array Descriptor Block (with limit checking field)





Array Descriptor Block (without limit checking field)

The format of the Descriptor Header in the Variable Descriptor must be as described in Paragraph 4.6.3.2 with the Data Type field set to 1101. The actual data within the array is accessed through the addressing information of the descriptor block. Note that the attached mode is not valid for the array structure since the actual array data can not immediately follow the descriptor header.

The first item in the auxiliary data is the Element Length. This field specifies the number of bytes in each element of the array. This information must be compatible with the information contained in the element descriptor to assure proper accessing of the data in the array. The Maximum Index field is used to check that the index contained in the instruction does not exceed the bounds of the array. This check will only be performed if the Flag field is set to a logic 1. The last field in the descriptor is the array element descriptor. This is a normal Variable Descriptor which specifies the format of the array element. In general, all of the access modes, data types, etc. are permitted in this descriptor. However, the operation of the address accessing modes is different from that described above for the normal Variable Descriptor (i.e. the attached mode becomes attached relocatable and short and long relative becomes short and long relocatable respectively) This is specifically designed to facilitate the direct support of complex data structures (multi-dimensional arrays, arrays of records, etc.). If the logical addressing access mode is specified, the specified logical address replaces the previous address calculated.

The sequence of operations which the MCS65E4 goes through to access a raw data item within an array structure is as follows (refer to the flow diagram below throughout the discussion - the sequence numbers correspond to the diagram):

1. The processor accesses the array descriptor block (ADB) through any of the MCS65E4 addressing mode. Set Y = the address following the addressing information.

2. A check is made to determine whether the descriptor indicates an array structure. If it is not, the processor continues to determine the desecriptor's type.

3. If the descriptor indicates array structure, the MCS65E4 gets the addressing information from the descriptor block.

4. The addressing information and the access mode bits together determine the starting address of the raw data for the first element of the array (In the flow diagram, this calculated address is saved in variable Y - an internal register. For attached mode, Y = Y + 0. For relocatable mode, Y = Y + relative. For logical mode, Y = the physical address.)). Note that the attached mode is not valid for an array structure since the actual raw data does not follow the descriptor header.

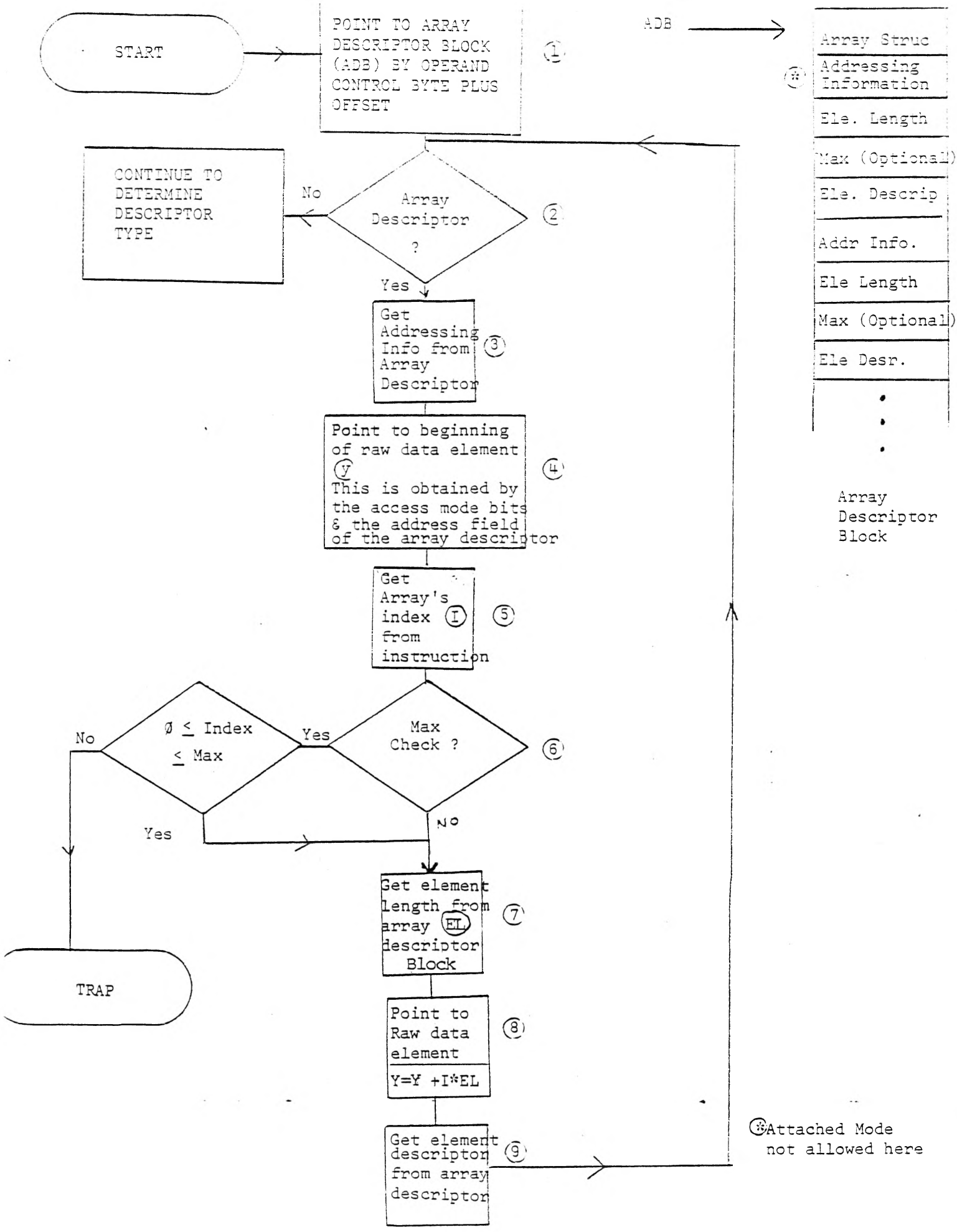
5. To determine which element in the array to access, the processor fetches the data specified by the next operand in the instruction. This data is treated as the index (In the flow diagram this value is referred as variable I).

6. However, before the element is accessed, the MCS65E4 determines if the index value is greater than or equal to zero and less than or equal to the maximum index value contained in the array descriptor. The check will not be performed if FLAG equals zero (see section 4.6.3.2.3.4). If FLAG equals one in the descriptor header and the index value is invalid, the processor terminates execution of the current instruction and will execute a trap sequence.

7. The processor then sets the element length from the descriptor block. The element length (in the flow diagram, this value is refered to as variable EL) within the array descriptor is the number of bytes of each element in the array.

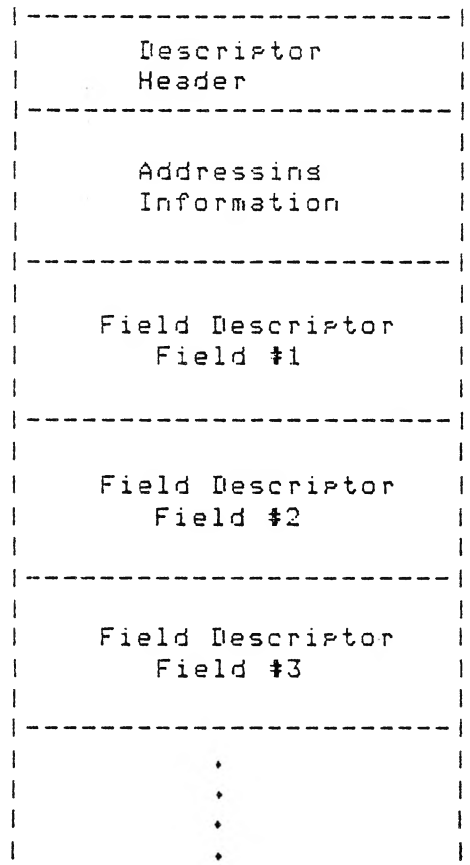
8. The element length (EL), the index value (I), and the previous calculated address (Y) are used to address the sought after element within the array structure. The formula for the address calculation is Y = Y + (I * EL). Note that the location of this element can be determined without regard for the exact nature of the data being stored in the array.

9. If the element descriptor specifies a basic data type (byte through string), the processor can process its address information to get the address of the raw data. Once the processor has the address of the raw data element, it can proceed to move the data into the chip. If however, the element discriptor is an array structure, the above sequence (3 through 9) will repeat. When a basic data element is encountered, only 3 and 4 will be performed to fetch the data field. In both cases, 4 has a different meaning since Y now contains the address to the base of the array. Therfore, attached mode (in this case called the attached relocatable mode) is used to access the first element of the array. Relative mode (in this case called short or long relocatable mode) now contains the the offset from previously calculated address instead of that of the descriptor.


START
POINT TO ARRAY DESCRIPTOR BLOCK (ADB) BY OPERAND CONTROL BYTE PLUS OFFSET
1
ADB
Array Struc
Addressing Information
Ele. Length
Max (Optional)
Ele. Descrip
Addr Info.
Ele Length
Max (Optional)
Ele Desr.
Array Descriptor Block
CONTINUE TO DETERMINE DESCRIPTOR TYPE
No
Array Descriptor ?
2
Yes
Get Addressing Info from Array Descriptor
3
Point to beginning of raw data element (Y) This is obtained by the access mode bits & the address field of the array descriptor
4
Get Array's index (I) from instruction
5
No
Ø ≤ Index ≤ Max
Yes
Max Check ?
6
Yes
NO
Get element length from array descriptor Block (EL)
7
TRAP
Point to Raw data element
Y=Y +I*EL
8
Get element descripton from array descripton
9
*Attached Mode not allowed here

4.6.4.4 Record

A record consists of a number of related but dissimilar data fields which are organized into a single data structure. When the MCS65E4 encounters a record descriptor during an instruction execution, it fetches the next byte of data from the instruction. This is assumed to be an offset from the first field descriptor to a field descriptor. The field descriptor then provides the information required to access the raw data. The format of the Record Descriptor is as follows:



As with all Variable Descriptors, the record descriptor begins with a Header. The Data Type field must be 1100 (binary) and the flag must be a logic 1. The Access Mode field can be either of the relative modes or the logical addressing mode. The attached Access Mode is not supported for the same reason mentioned above for the array structure. Therefore, the addressing information is not optional in the Record Descriptor.

The format of the Field Descriptor is exactly like that of the normal Variable Descriptor. However, the operation of the offset and logical addressing access modes is different from that described above for the normal Variable Descriptor and similar to that described for the Variable Descriptor. This is specifically designed to facilitate the direct support of complex data

structures (multi-dimensional arrays, arrays of records, etc.). If the short or long relocatable offset is specified in the field descriptor, the offset will be added to the previous address calculated. If the attached relocatable addressing mode is specified, then zero is added to the previous address calculated. If the logical addressing access mode is specified, the specified logical address replaces the previous address calculated. This is illustrated in detail in the examples below.

The sequence of operations which the MCS65E4 goes through to access a raw data item within a record structure is as follows (refer to the flow diagram below throughout the discussion - the sequence numbers correspond to the diagram):

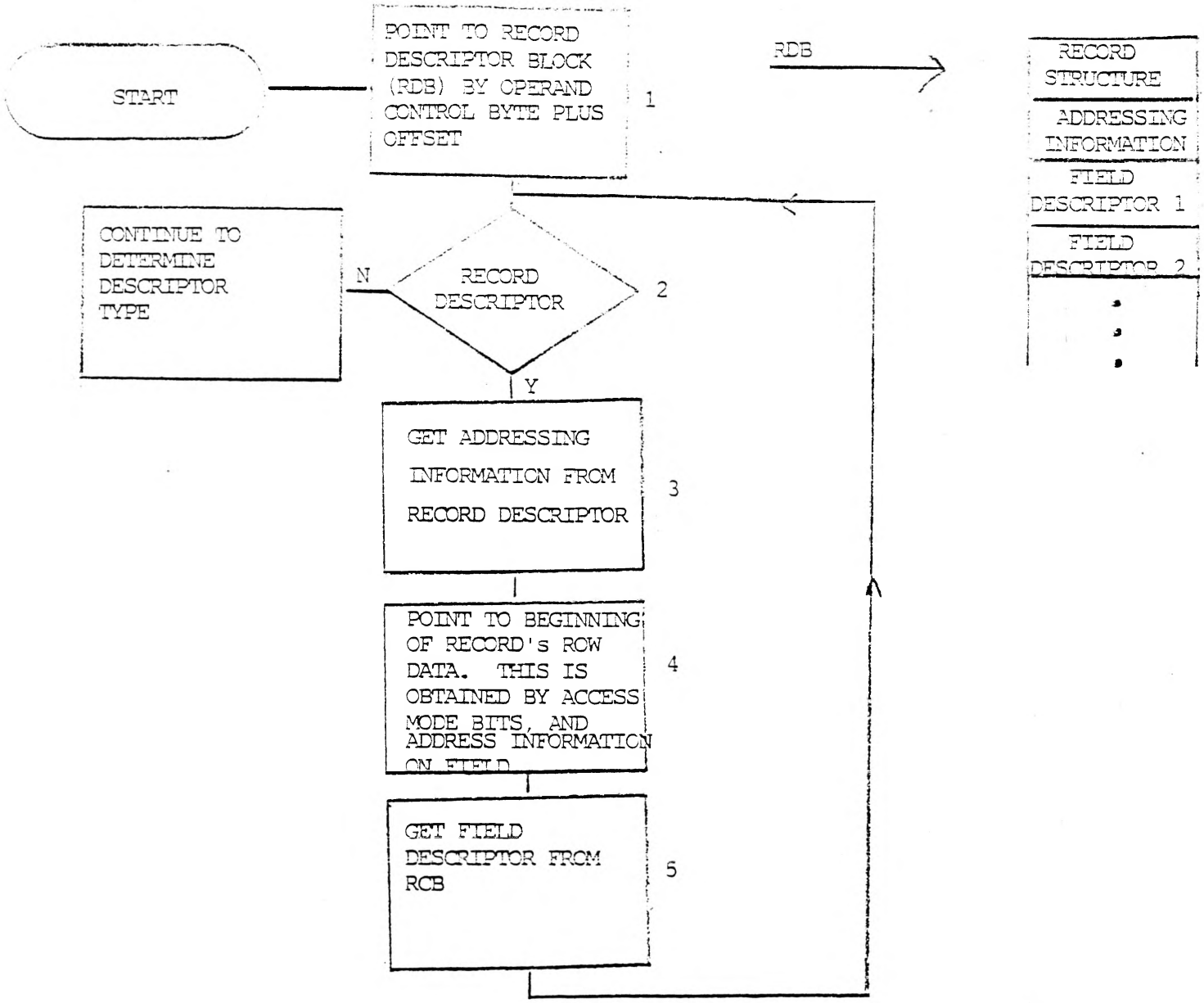
1. The processor accesses the record descriptor block (RDB) through any of the MCS65E4 addressing modes.

2. A check is made to determine whether the descriptor indicates a record structure. If it is not, the processor continues to determine the desecriptor's type.

3. If the descriptor indicates array structure, the MCS65E4 gets the addressing information from the descriptor block.

4. The addressing information and the access mode bits together determine the starting address of the raw data for the first field. In the flow diagram, this calculated address is saved in variable Y - an internal register. For attached mode, Y = Y + 0. For relocatable mode, Y = Y + relative. For logical mode, Y = the physical address.

5. To determine which field within the record to access, the processor fetches the data specified by the next operand in the instruction. This data is treated as the offset into the record descriptor to obtain the field descriptor.

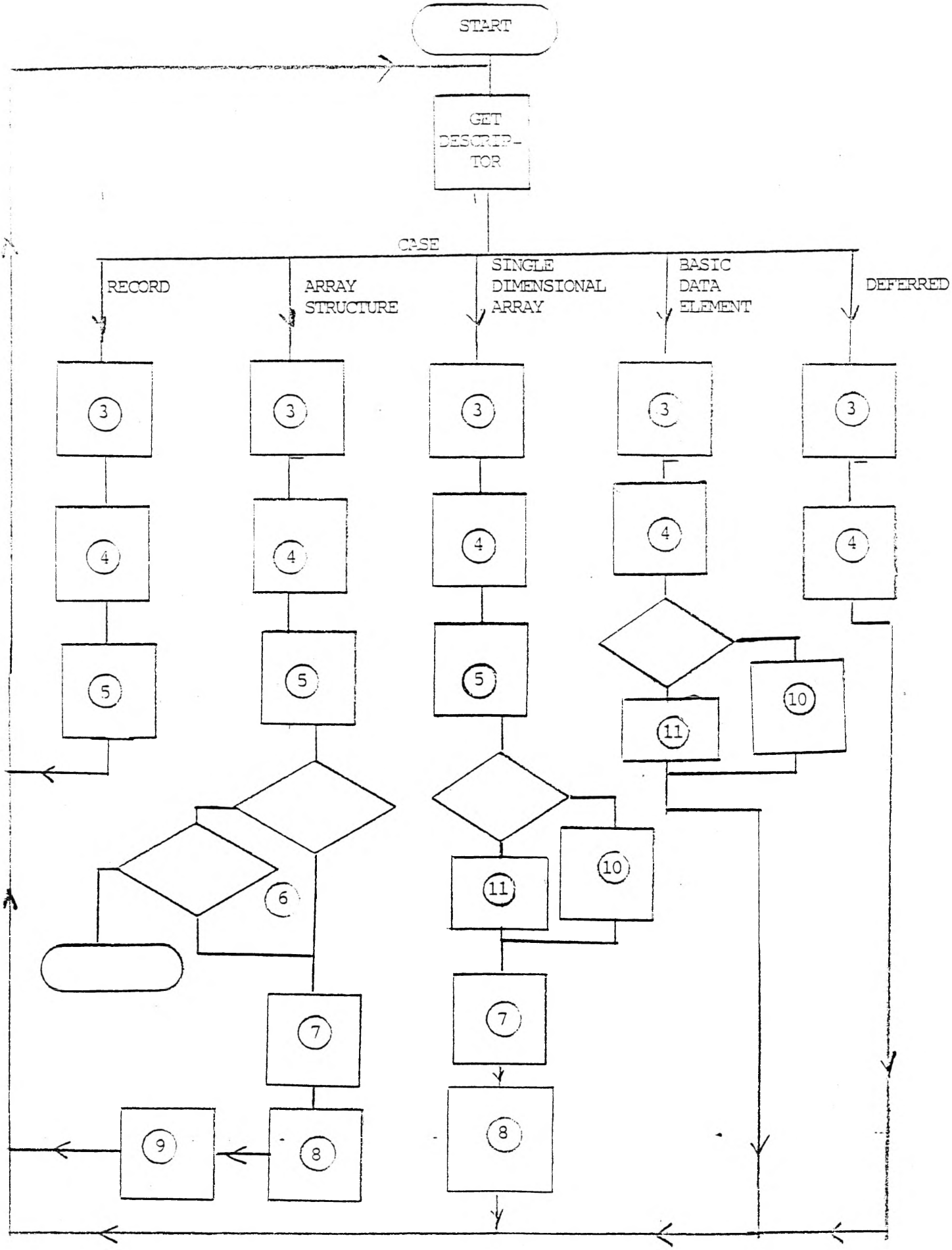

START
POINT TO RECORD DESCRIPTOR BLOCK (RDB) BY OPERAND CONTROL BYTE PLUS OFFSET
1
RDB
RECORD STRUCTURE
ADDRESSING INFORMATION
FIELD DESCRIPTOR 1
FIELD DESCRIPTOR 2
CONTINUE TO DETERMINE DESCRIPTOR TYPE
N
RECORD DESCRIPTOR
2
Y
GET ADDRESSING INFORMATION FROM RECORD DESCRIPTOR
3
POINT TO BEGINNING OF RECORD's ROW DATA. THIS IS OBTAINED BY ACCESS MODE BITS, AND ADDRESS INFORMATION ON FIELD
4
GET FIELD DESCRIPTOR FROM RCB
5

This page intentionally left blank

4.6.5 Deferred Descriptor

The deferred descriptor contains no information regarding the type of data which is stored in the data field which is to be accessed by the instruction. For this reason, the deferred descriptor does not directly refer to the raw data. Instead, it points to another descriptor which can contain data type information. It should be noted that this second descriptor is exactly the same as any variable descriptor (i. e., the second descriptor could very well be another deferred descriptor). In all cases, it is necessary that the processor encounter data type information before raw data can be fetched from memory. The operation of the deferred descriptor is illustrated in Paragraph 4.6.6.4.

A summary of all the data types is depicted below:


START
GET DESCRIP-TOR
CASE
RECORD
ARRAY STRUCTURE
SINGLE DIMENSIONAL ARRAY
BASIC DATA ELEMENT
DEFERRED
3
4
5
6
7
8
9
10
11

4.6.6 Application of the MCS65E4 Data Accessing Mechanisms

4.6.6.1 Introduction

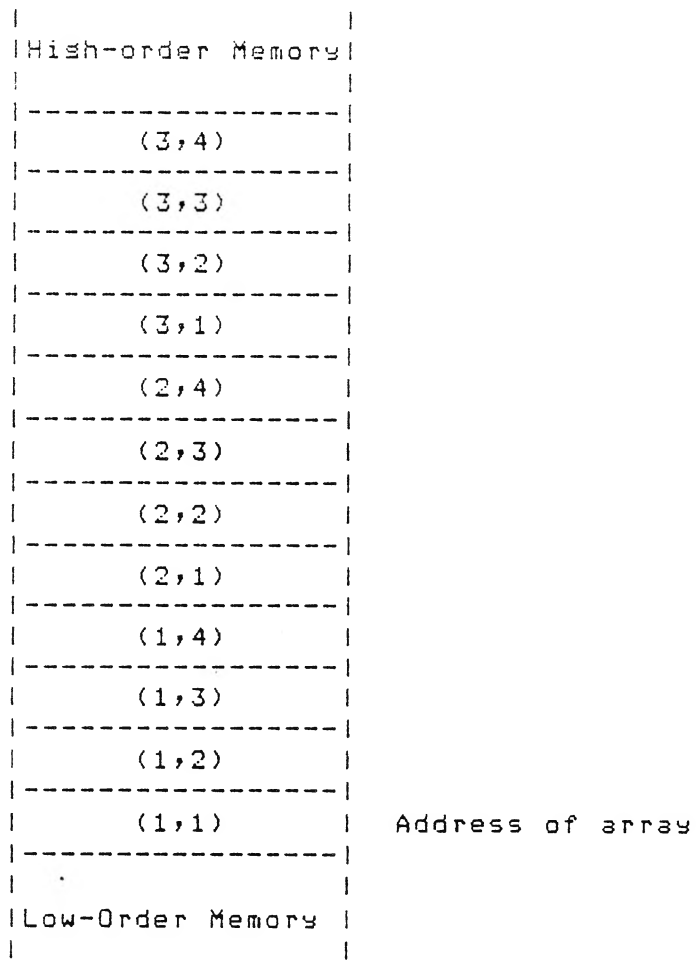
The data accessing mechanisms described above are used throughout the MCS65E4 architecture for storing raw data, array indexes, exception vector addresses, etc. For this reason, they are a very important key to understanding the operation of this processor. Therefore, a number of examples of data structures are described in detail below. It is hoped that these will contribute to the readers understanding of the manner in which data is stored and accessed within the MCS65E4 system.

4.6.6.2 Accessing Data in Multi-Dimensional Array Structures

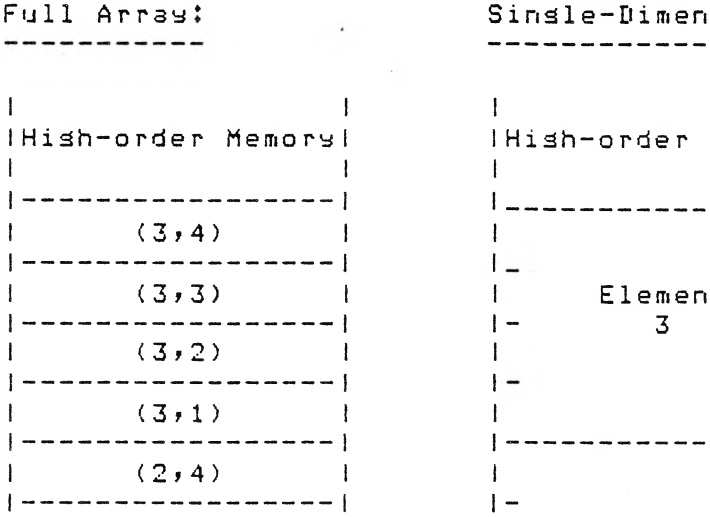
The manner in which the data storage and accessing techniques described above can be extended to control the accessing of data in complex array structures can be described most effectively by reviewing the sequence of operations which takes place when the MCS65E4 encounters the Array Descriptor. As described previously, the addressing information contained in the descriptor specifies the address of the first element in the array being accessed. The element length information contained in the descriptor is then combined with the next operand in the instruction to determine the location of the array element being accessed.

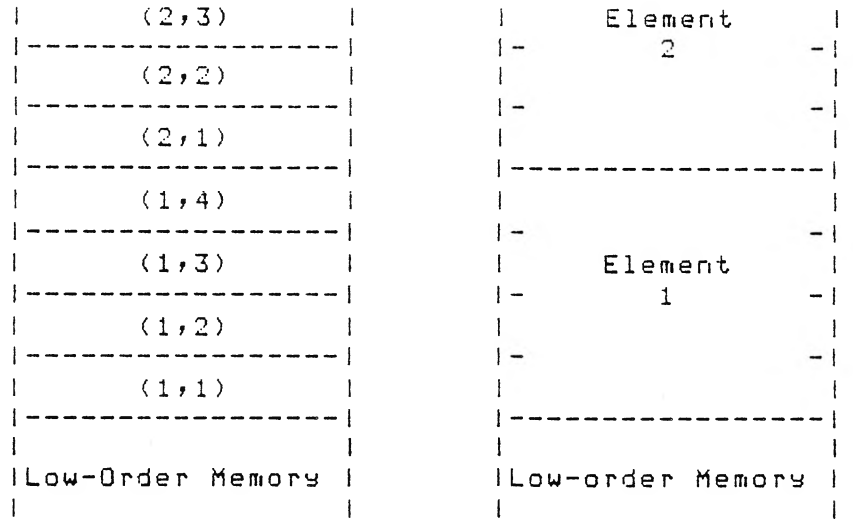
After the physical address of the array element has been determined, the processor then references the element descriptor to determine the manner in which the element is to be processed. As noted above, the format for the element descriptor is essentially the same as for any of the normal descriptors. Specifically, this means that the element descriptor can specify any of the normal data types such as the scaler (byte, ordinal, real, etc.). Even more important is the fact that the element descriptor can specify that the element is a simple array, an array structure, or a record. This is the key to accessing complex data structures in the MCS65E4 system.

The operation of the element descriptor can be illustrated by examining the manner in which the MCS65E4 would access data in a simple 3 X 4 two- dimensional array (termed DATA(X,Y) below). The twelve elements in this array would be stored in contiguous memory locations as follows:



To create the descriptor for this array, the structure must be viewed as a single-dimension array in which each element contains four data fields. This can be illustrated as follows:





The descriptor which specifies the form of this single-dimension array must contain an element length which is four times the actual element size. For example, if the raw data is ten byte real then the element length in the array descriptor would be forty.If the index is to be checked for maximum size, the descriptor must specify 3 as the maximum for the first index into the array. Using the procedures described above, the MCS65E4 will first utilize the addressing, element size and index information to determine the location of an element in this single-dimension array. For instance, if the instruction references element (2,3) in the array, the processor will first determine the address of element 2 in the single-dimension array by evaluating the following formula:

Physical Address = (Contents of Specified Base Register)
+ (Array Address) + (Element Size * Index)

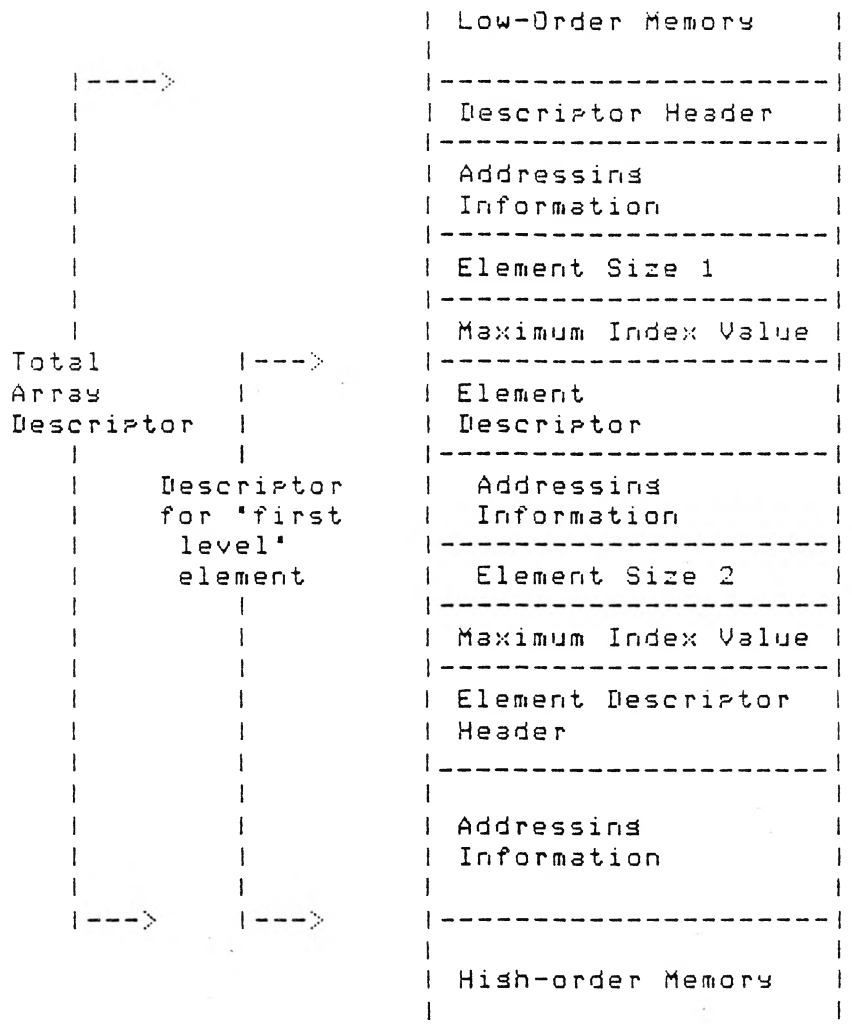
Note: * = multiplication

After this operation is complete, the MCS65E4 begins processing the element descriptor. This operation is very similar to normal descriptor processing except that the Relative and Attached addressing becomes short and long relocatable as required for accessing data in the array structures. Rather than adding the displacement to the address of the descriptor, the processor instead adds it to the address which was determined during the previous phase of the addressing sequence in progress. In the example above (accessing element (2,3) in the array), the address determined during the first phase of the addressing sequence would be that of element (2,1) in the array. This corresponds to element 2 in the single-dimension depiction illustrated above.

For the 3 X 4 two-dimensional array described previously, there are two methods which could be utilized for specifying the element descriptor. If each element in the total array is a scalar and if no index checking is required, the element descriptor can specify that each element in the single-dimension array is a simple scaler array. If index checking is desired, then the element descriptor must specify that each element of the single-dimension array is an

array structure. This element descriptor would then contain a second element descriptor which would specify the exact data type for the array element.

The above discussion becomes somewhat confusing because of the "levels" which the processor must go through during the addressing sequence. This can be illustrated more clearly by continuing the example from above. Most importantly, the reader must understand the fact that during the initial processing of the descriptor, the MCS65E4 views this two-dimensional array as a single-dimension array structure. This "first-level" of addressing allows the processor to access array elements in sets of four (for example, (2,1), (2,2), (2,3) and (2,4)). The element descriptor within this array descriptor must therefore contain the information required for accessing the "second level", i.e., for determining the exact location of the desired element within the groups which are accessed during the first level. The total descriptor for the example above (assuming index checking is required and that each element of the array contains a byte data type) would therefore be:



There are several important points which can be noted from this illustration. The element descriptor's addressing information in the previous example can be eliminated since attached addressing mode (in this case called attached relocatable mode) is utilized. The physical address of (2,3) equals:

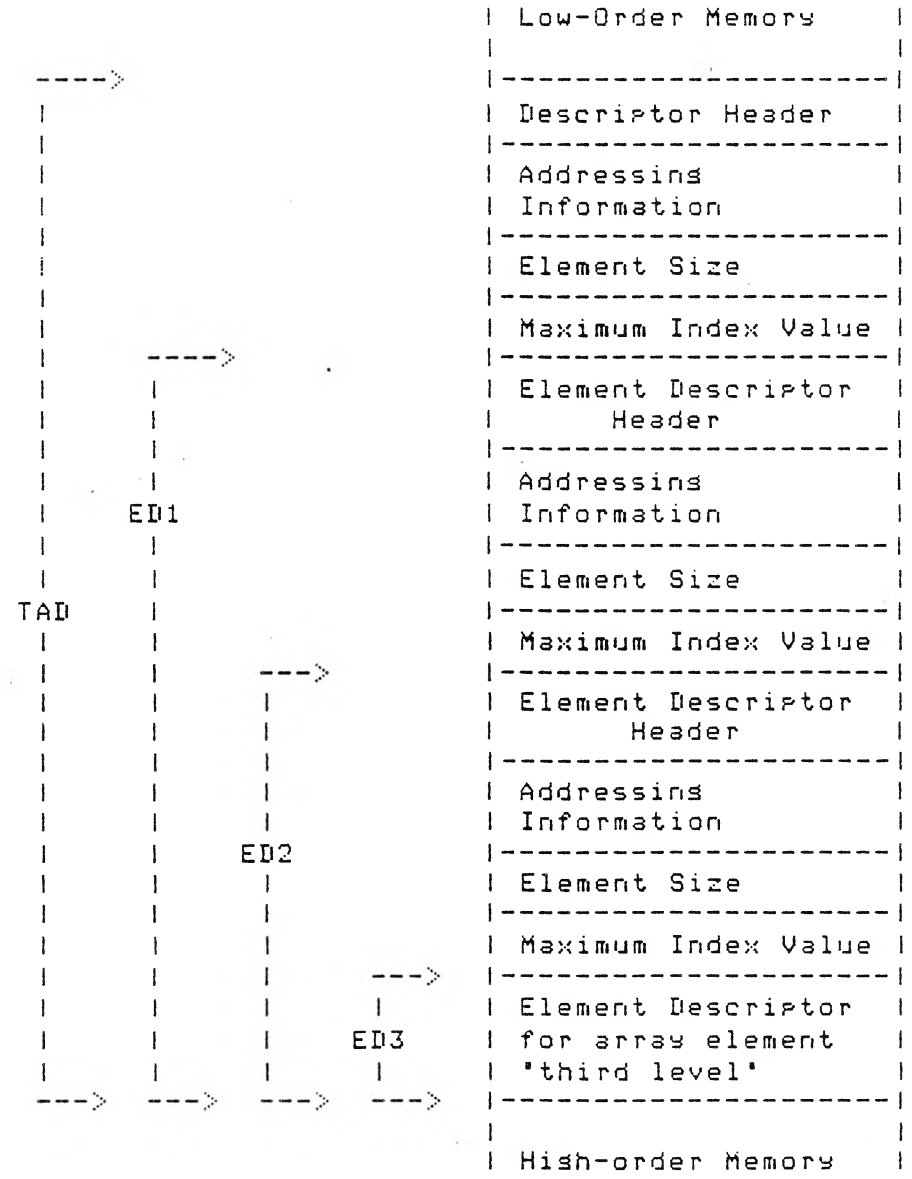
(contents of specified base register) +
(base of array logical address) +
(element size 1 * index 1) +
(element 2 addressing information) +
(element size 2 * index) +
Data Addressing Information

If this is a 3 X 4 array of 10 byte Reals, the address is calculated as follows:

Data Address = Physical Address of the base of array +
(40 * 1) + 0 + (10 * 2) + 0

= Physical address of the base of array + 60

Note that index 1 = 2 - 1 = 1; index 2 = 3 - 1 = 2. The element addressing information equals 0 for attached mode and offset for short or long relocatable. There is no way for the processor to determine the total length of the array descriptor. This is the reason that attached addressing is not valid in the first Descriptor Header. Also, it should be noted that the nesting of the second level descriptor into the first level descriptor can be extended to any level of complexity. This means that the second element descriptor shown in this diagram can, in fact, be another array descriptor or even a record descriptor. In all cases, however, care must be taken to assure that the element length be specified properly and that all addressing be specified in a manner which allows the processor to process each level of addressing.



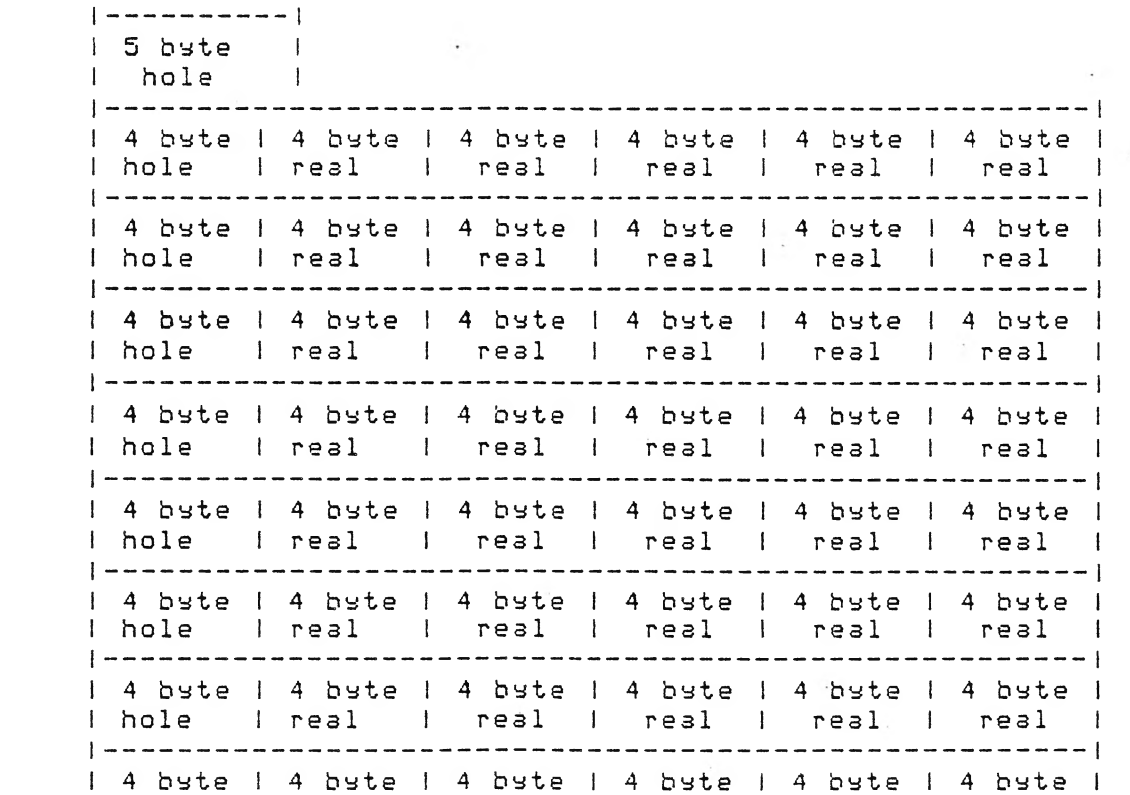
TAD : Total Array Descriptor
ED1 : Element Descriptor For First Level Element
ED2 : Element Descriptor For Second Level Element
ED3 : Element Descriptor For Third Level Element

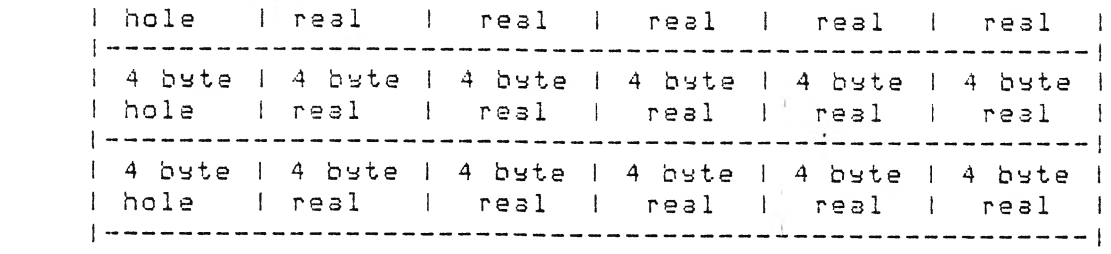
4.6.6.3 Example of Accessing Multi-Dimensional Array

For our next example we assume we have a FORTRAN data area where data within the area can be accessed either as single eight-byte integers or single four-byte reals through an equivalence statement. The data in memory has 'holes' in it (i.e. the real or integers are not contiguous). However, the MCS65E4 descriptor can be setup to be able to access the data properly (i.e. to by-pass the 'holes'). Assume we have a three dimensional array, A, with maximum dimension [15,10,5]. The data is stored sequentially as:

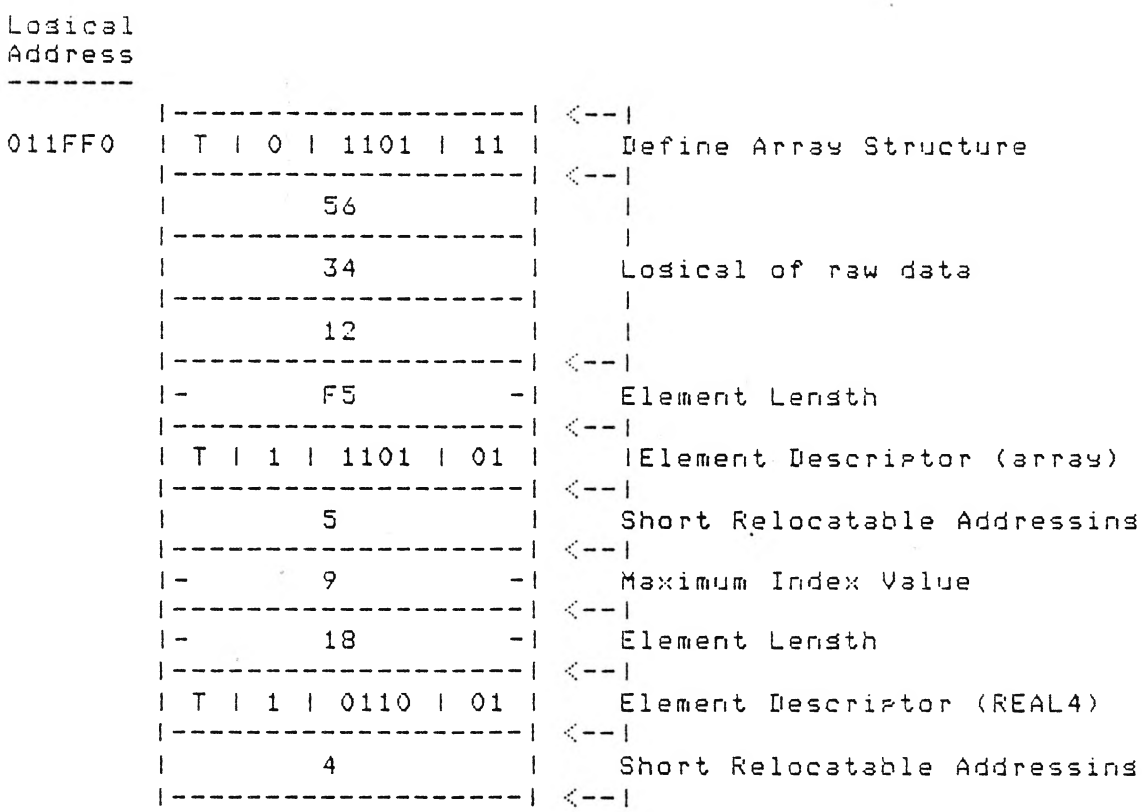
```
      A(0,0,0), A(0,0,1), ..., A(0,0,4)
      A(0,1,0), A(0,2,0), ..., A(0,9,4)
      A(1,0,0), A(2,0,0), ..., A(9,0,0)
      .         .             .
      .         .             .
      .         .             .
                              A(14,9,4) (without showing 'holes')
```

Geometrically, the array consists of fifteen planes where each plane consists of one 10x5 matrix plus some imbedded 'holes'. The plane is depicted below. Remember there are fifteen of these planes to form a solid geometric structure.





The descriptor for this array is shown below with a description of how each field is derived.



Descriptor for Multi-Dimensional Array

Field	Description
1.	Specifies first dimension as array structure with logical addressing.
2.	Specifies the logical address of the array (123456)
3.	Specifies the element length for first dimension. Element length = H1 + MAX2 * (H2 + (MAX3*L)) (where H1 = hole 1, MAX2 = second dimension size, H2 = hole 2, MAX3 = third dimension size, L = length of element)

Element length = 5 + 10 * (4 + (5*4))
= F5 (hex) using above formula:

4. Specifies second dimension as array structure with short relocatable mode addressing

5. Specifies the five byte offset (i.e. skip 5 byte hole at beginning of each plane).

6. Specifies maximum index value for second dimension

7. Specifies element length for second dimension.
Element length = 4 + 5*4
= 18 (hex) using formula:

Element length = H1 + MAX3*L

(where H1 = 4 hole
MAX3 = third dimension size)

8. Specifies the third dimension as an array of four-byte reals (this as accomplished by setting FLAG = 1) with short relocatable offset mode addressing.

9. Specifies the four byte offset (skip over the hole).

The processor will access A[10,8,3]. This assembles into:

PRM + 1 Ext Byte	10	#0A	#08	#03

The contents of the PRM register is 011FE0. To set to the address of the array descriptor the processor will add the contents of the primary register (011FE0) with the next byte in the instuction (10) giving 011FF0. The next byte (#0A) is an operand control byte and will be used by the processor to calculate the first index of the sought after element and #08 for the second index and #02 for the third. Using the array descriptor, the processor calculates the address of A[10,8,2] with the following formula:

```
Q = AID + (EL1 * I1) +
          E2A +
          (EL2 * I2) +
          E3A +
          (L * I3)
```

where AID = Address Information of Descriptor
EL1 = Element 1 Length
I1 = Index 1
E2A = Element 2 Addressing Information (relocatable)

```
EL2 = Element 2 Length
I2  = Index 2
E3A = Element 3 Addressing Information (relocatable)
L   = Data Length
I3  = Index 3
```

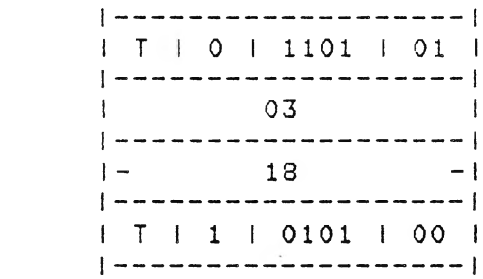
Substituting the example data into the above formula we get:

```
Q = 123456 + (F5 * 0A) +
             5 +
             (18 * 08) +
             4 +
             (4 * 3)
Q = 123EBD
```

At logical address 123EBD is the data for element A[10,8,3]

If we access element A[I,J,K] where I=19, J=5, and K= -8, the processor will not trap out for 'element-out-of-range' since both I and K do not have range checking specified in the desrciptor. However when I=0, J=10, and K=1 the processor will trap out when J is processed since J is greater then 9 (i.e. 9 is the max index as specified in descriptor).

This FORTRAN data area can also be accessed as a two dimensional array of eight-byte integers through an equivalence statement. The array, B, has maximum dimension 10X3. Each row of data is now treated as three eight-byte integers instead of five four-byte reals. Note that the four byte hole at the beginning of each row will now be utilized. The descriptor for array B will be as follows:



Field	Definition
1.	Specifies array structure with short relocatable addressing mode.
2.	Specifies the short relocatable offset
3.	Specifies the element length of the first dimension. Element length = MAX2*L = 3 * 8 = 18 (hex)

4. Specifies an array of 8-byte integers with attached addressing mode.

Note that the descriptor for array B is five bytes long which will fit into the five byte "hole" in the beginning of each plane. If we use external base addressing we can modify the base page and address any plane within the array. For example to access B[8,2]:

External Base with no offset	20	$08	$02

The Original 20 into the BAS page is 123DE8 which is the address of variable B. Note that this by-passes the first dimension of the previous example (A[10,8,3]). Using the above formula we can address A[10,8,3] by specifying B[8,2]:

```
Q = 123DEA+    (see note below)
    3+
    18*08+
    02*8+
  = 123EBD
```

note : The relative calculation will start from the byte following the address information.

Note that element A[10,8,3] is referring to element B[8,2] (i.e. both generated addresses are equal). This is shown below:

hole	0	1	2	3	4	Third Element of Array A

(both have same address) =

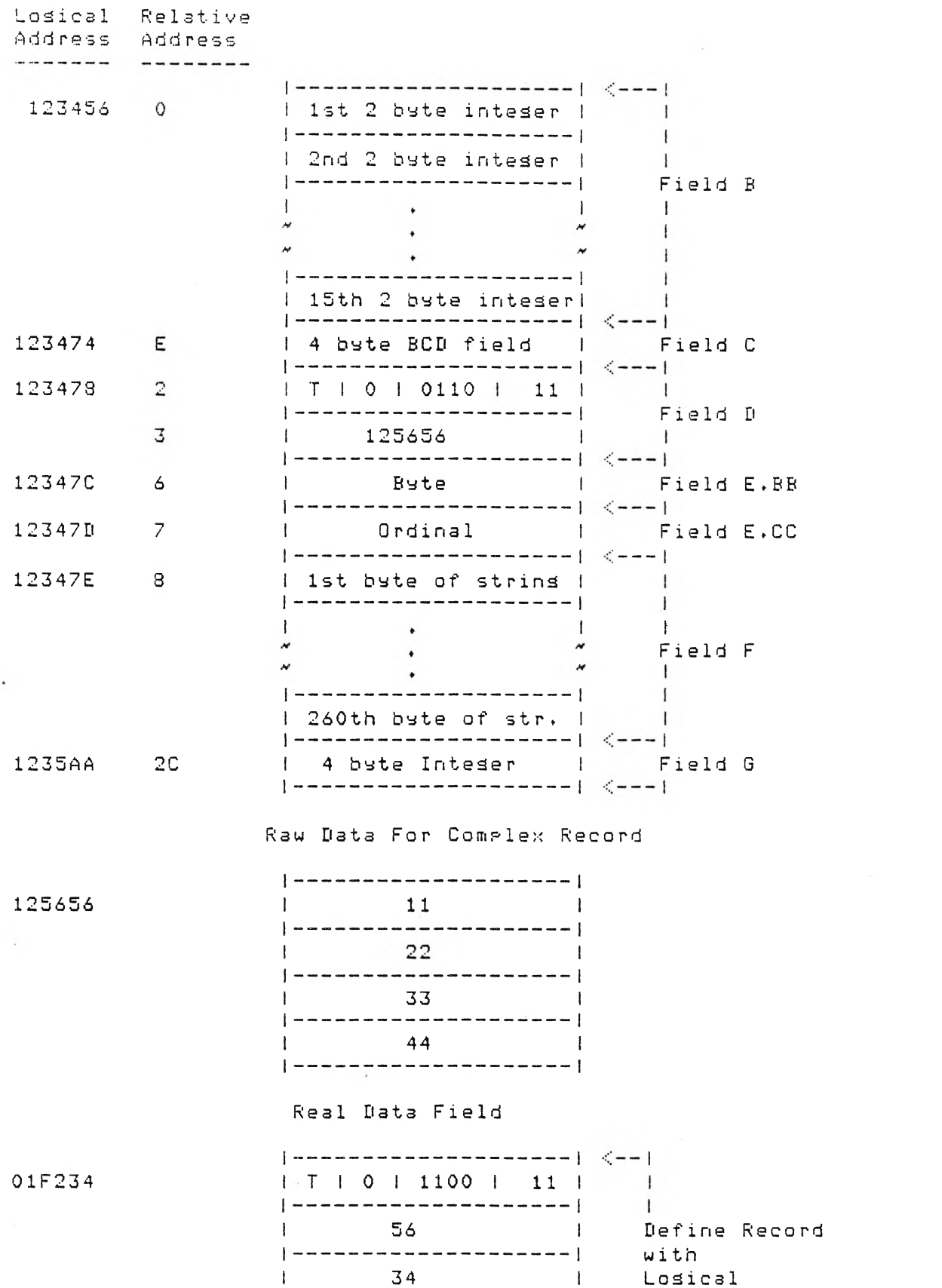
0	1	2	Second Element of Array B

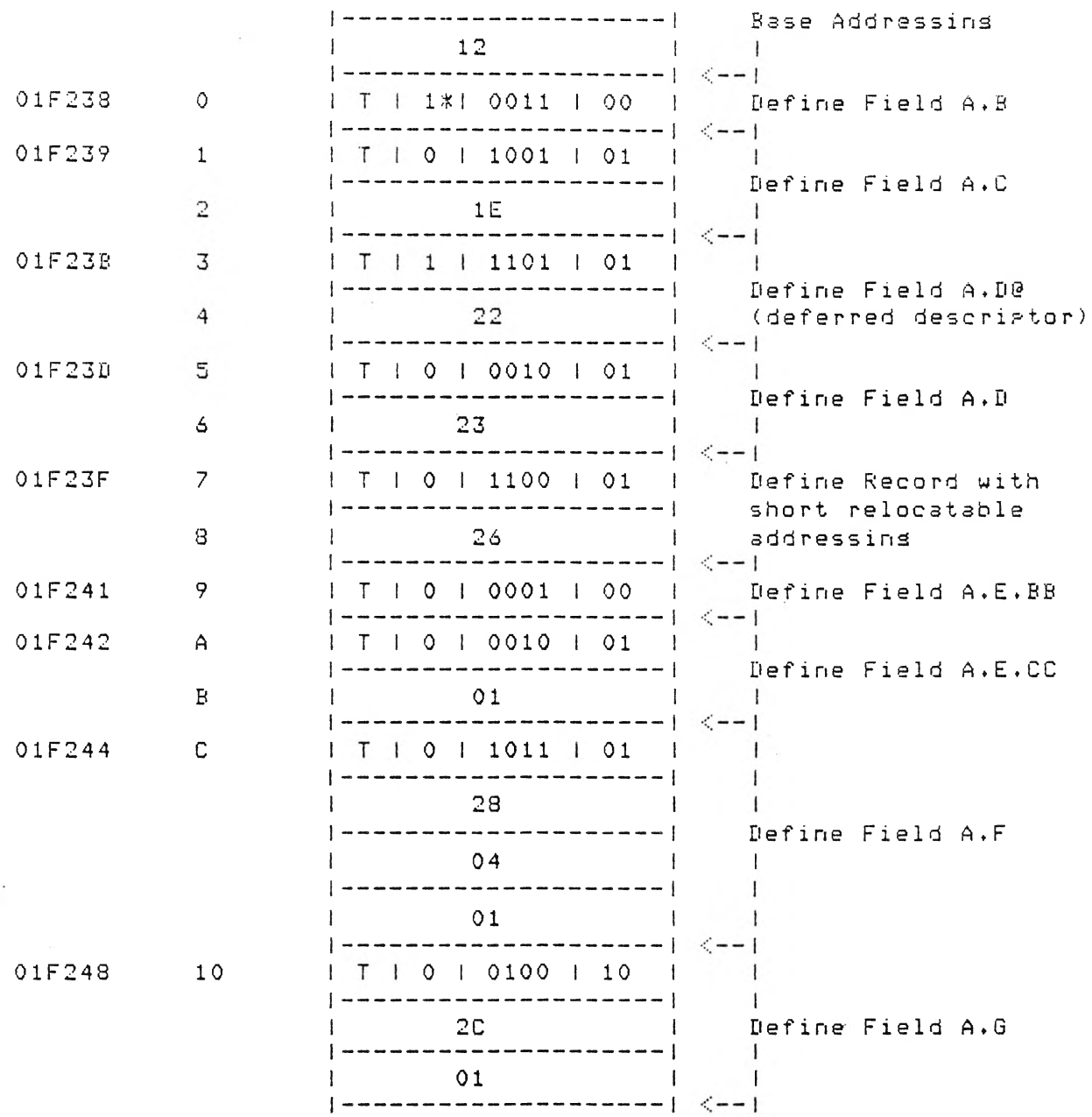
4.6.6.4 Example of Accessing Data in a Complex Record Structure

The complex record structure can best be described by referring to a diagram and following the operations the processor must perform to access the data. Assume we have a PASCAL-like record shown below:

```
A = Record
    B : ARRAY [1..15] of INTEGER*2; {define 2 byte integer array}
    C : BCD4;                       {define 4 byte BCD field}
    D : REAL4@;                     {define pointer to 4 byte REAL}
    E = Record                      {define new record}
        BB : BYTE;                  {define a byte field}
        CC : ORDINAL                {define an ordinal field}
      End;                          {end record}
    F : ARRAY [1..260] of CHAR;     {define a 260 byte string}
    G : INTEGER*4                   {define a 4 byte integer field}
      End;                          {end record}
```

As previously described a record is a group of related but dissimilar data fields which are organized into a single data structure. The descriptor for an MCS65E4 record consists of a record header, addressing information, and field descriptors. Basically, the header defines the data type as record, the addressing information sets the processor to the start of the raw data and the field descriptors define the specific fields within the record. There is a one-to-one correspondence between a field descriptor and the data field in the record. Within the field descriptor is addressing information which is the offset from the base of the record. The base address of the raw data (contained in the record descriptor) and the offset (contained in the field descriptor) provide the processor with sufficient information to address any field within the record. The following example illustrates accessing a complex record by depicting a record's descriptor and raw data:

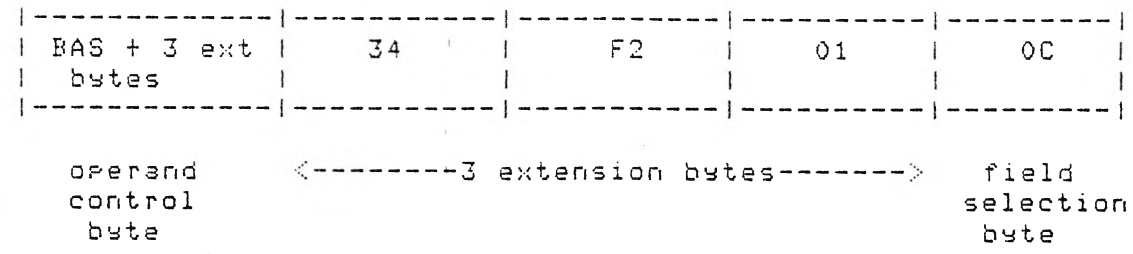

Logical Address
Relative Address
123456
0
1st 2 byte integer
2nd 2 byte integer
15th 2 byte integer
Field B
123474
E
4 byte BCD field
Field C
123478
2
T | 0 | 0110 | 11
Field D
3
125656
12347C
6
Byte
Field E.BB
12347D
7
Ordinal
Field E.CC
12347E
8
1st byte of string
260th byte of str.
Field F
1235AA
2C
4 byte Integer
Field G
Raw Data For Complex Record
125656
11
22
33
44
Real Data Field
01F234
T | 0 | 1100 | 11
56
34
Define Record with Logical



* - Note: Single Dimension Array

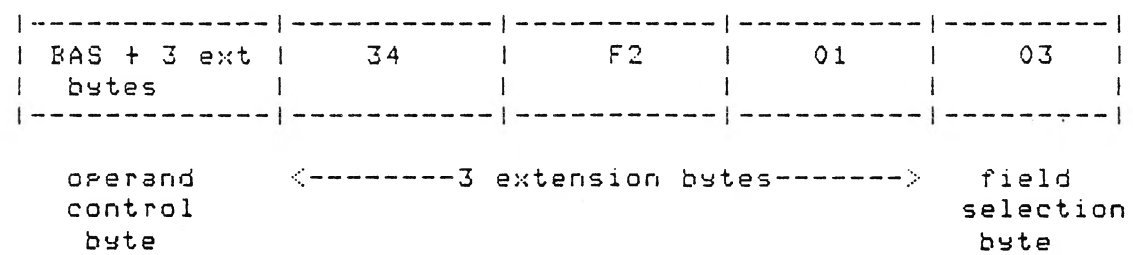
Descriptor For Complex Record

The field A.F is to be accessed. The instruction assembles into:



The operand control byte followed by three extension bytes will point to the descriptor at logical address 01F234 (assume BAS register equals 0). After determining that the descriptor is logical addressing mode the processor uses the next three bytes (starting at 01F235) to point to the start of the raw data for the record. Since the descriptor specifies that it is record data type, the MCS65E4 then fetches the next byte from the instruction (0C in our example) and uses this as a displacement into the record descriptor to obtain the appropiate field descriptor. In our example this is located at 01F244 (01F238 + 0C). The field descriptor header indicates string data type with short relocatable offset. The offset in our example is 28 which is located after the descriptor header (01F245). This offset is then added to the start of the raw data (123456 + 28 = 12347E) which is the address of the first byte of the string. The next two bytes of the field descriptor is the string length.

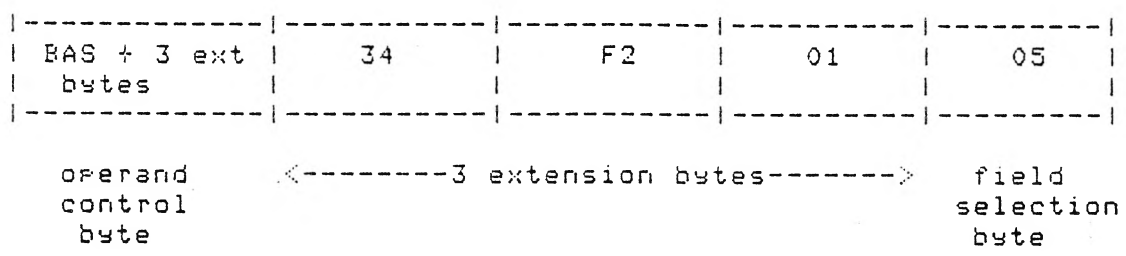
As another example, the field A.D@ is to be accessed. The instruction assembles into:



The operand control byte followed by three extension bytes will point to the descriptor at logical address 01F234 (assume BAS register equals 0). After determining that the descriptor is logical addressing mode the processor uses the next three bytes (starting at 01F235) to point to the start of the raw data for the record. Since the descriptor specifies that it is a record data type, the MCS65E4 then fetches the next field selection byte from the instruction (03 in our example) and uses this as a displacement into the record descriptor to obtain the appropiate field descriptor. In our example this is located at 01F23B (01F238 + 03). The field descriptor header indicates deferred type with short relocatable offset. The offset, 22 in our example, is added to 123456 giving 123478. Since this a deferred descriptor, the data contained at 123478 is not raw data but another descriptor. The descriptor, located at 123478, is then processed the same as

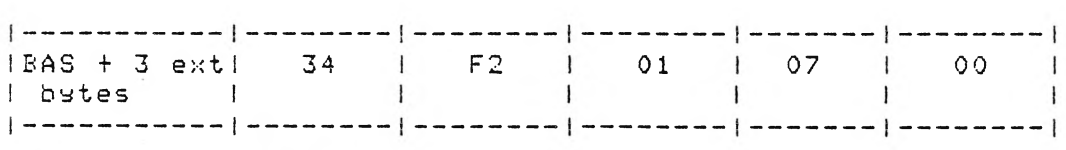
any other descriptor. In our example, the descriptor indicates four-byte real type with logical base addressing (i.e. the data is located at 125656). Therefore, the actual raw data is 11223344 (see diagram above).

Similarly, the field A.D can be accessed. The instruction assembles into:



The A.D differs from A.D@ in that the former refers to the logical address that points to the four-byte real. Therefore the field descriptor indicates Ordinal data type with short relocatable offset. The offset is 23 (one more byte than A.D@), so it points to the logical address instead of the four-byte real descriptor. Using this technique we can implement any pointer data, such that P is a pointer to pointer to pointer to the four-byte real, i.e., P,P@, P@@, P@@@. All have meaning and can all be implemented by the MCS65E4 data structure.

For the last example, field A.E.BB is to be accessed. The instruction assembles into:



The operand control byte followed by three extension bytes will point to the descriptor at logical address 01F234 (assume BAS register equals 0). After determining that the descriptor is logical addressing mode the processor uses the next three bytes (starting at 01F235) to point to the start of the raw data for the record. Since the descriptor specifies that it is a record data type, the MCS65E4 then fetches the next field select byte from the instruction (07 in our example) and uses this as a displacement into the record descriptor to obtain the appropriate field descriptor. In our example this is located at 01F23F (01F238 + 07). The field descriptor header indicates record type with short relocatable offset. The offset, 26 in our example, is added to 123456 giving 12347C. Since this is another record, the processor must get another byte from the instruction to determine the offset from the base of the record descriptor. In our example, this byte is 0 which is added to 01F241 giving 01F241 which is the field

descriptor for A.E.BB. This field descriptor indicates byte data type with attached (relocatble) addressing mode (i.e. Add 0 to 12347C giving 12347C). Therefore, the raw data for A.E.BB is located at 12347C. The processor calculates the address of a field in the record by using the following formula:

(record address information) + (field descriptor 1 relocatable offset) + (field descriptor 2 relocatable offset)

4.6.6.5 Exception Vectors

4.6.6.5.1 Introduction

Each of the exception vectors listed in Section 4.4.4 is stored in a four - byte field in a manner which is compatible with the MCS65E4 data structure. The first byte of the vector contains a descriptor which primarily specifies that the logical address of the exception processing software is either 'attached' to the descriptor or is located in a 'remote' data structure. If the address is attached, the three bytes following the descriptor are assumed to be the logical address of the software which will service the exception (exeception handler). This address will be added to the contents of the Process Base Register to determine the corresponding physical address. This address is then transfered into the Process Program Counter. The processor then begins execution of the exception software. In addition to the simple 'attached' form of the exception vector, the logical address can be located in a data structure which is separate from the four byte exception vector. In this case, the three bytes following the descriptor contains the logical address of the data structure in which the address of the execution software is stored. In all cases this vector format must conform to the MCS65E4 data structure described above and the data field which is referenced by the vector must be a three byte ordinal. This ordinal is assumed to be the logical address of the exception handler.

The use of complex data structures within the exeption vector is particularly useful for interrupts, systems calls, etc. in which a byte of information is generated by the exeption to be used as an index value into the array. A good example of this is the System Call in which a byte of data is passed to the operating system. In addition to placing this 'exeception qualifier' onto the stack, the data is retained in an internal working register for use during the exception vector processing. This allows the use of an array of ordinals to implement a direct vectored system call, direct vectored interrupts, etc. Each of these options will be illustrated in the examples below. These 'exception qualifiers' are described in detail in the discussion of the exception vectors.

4.6.6.5.2 Descriptor Format

The descriptor and associated data within the exception vector must conform exactly to the format described above for the MCS65E4 data structure. The TRAP bit is used to indicate whether the

exception can be serviced within the current process (T = 0). If the T bit is a logical 1, the processor will return to the lower level process to service the exception. Only one basic data type (ordinal) is valid. If the data type is a single dimension ordinal array, only a single byte of additional information is available for use in determining the index value of the ordinal array. Therefore, the exception vector represents a limited sub-set of the data types which the MCS65E4 supports.

4.6.6.5.3 Example of Attached Address Exception Vector Format

The simplest form of exception vector is one in which the logical address of the software which will service the exception (exception handler) directly follows the descriptor. This is illustrated below for an Overflow Trap whose handler is located at logical address 0034A0. Note that the TRAP bit in the descriptor is set to logic 0 and that the MODE field contains 00, specifying attached data. The contents of the exception vector will be as follows:

Address	Contents	Remarks
E8*	T,0,0010,00	Descriptor
E9*	A0 (HEX)	Low order byte of exception handler address.
EA*	34 (HEX)	Middle order byte of exception handler address.
EB*	00 (HEX)	High order byte of exception handler address.
0034A0	Actual exception handler code for overflow condition	

* - Page address within LMT page

4.6.6.5.4 Example of Remote Exception Vector

In many processes, particularly within the Kernel and Operating System processes, it is very likely that the exception vectors will be located in Read-Only Memory. In this case, use of the attached form of the exception vector would not allow the vector address to be modified during process execution. However, use of the remote exception vector allows the vector address to be placed in READ/WRITE memory. This will be illustrated by describing a Data Access Trap Vector in which the address of the handler is located in logical addresses 000003 through 000005 within the process. The exception handler begins at logical address 01C450 within the process. In this case, address D8 within the Limit Page contains a descriptor which specifies a remote ordinal data field. The remaining bytes of the vector contain logical address 000003. This causes the processor to fetch the address of the exception handler from logical address 000003 through 00005.

Address	Contents	Remarks
D8*	T,0,0010,11	Descriptor
D9*	03 (HEX)	Low order byte of the pointer to the exception handler.
DA*	00 (HEX)	Middle order byte of the pointer to the exception handler.
DB*	00 (HEX)	High order byte of the pointer to the exception handler.
000003**	50 (HEX)	Low order byte of exception handler address.
000004**	C4 (HEX)	Middle order byte of exception handler address.
000005**	01 (HEX)	High order byte of exception handler address
01C450	Actual exception handler code for data access trap	

* - Page address within the LMT page
** - Logical address within the process

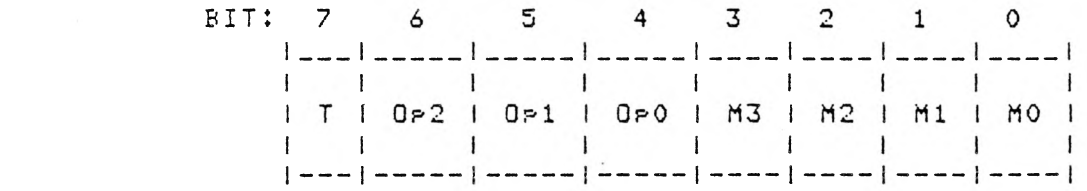
4.7 The MCS65E4 Instruction Set

4.7.1 Introduction

The use of descriptors to control accessing data within the MCS65E4 has a significant impact on the nature of the instruction set which is executed by this processor. Most importantly, it allows the use of "generic" instructions, i. e., instructions which do not specify those aspects of the operation which are data dependent. For example, like most processors, the MCS65E4 instruction set contains an arithmetic ADD instruction. However, this instruction only specifies that an add operation is to be performed between two data fields. The exact nature of the add operation is determined by the data contained in the data fields (BCD add for BCD data, etc). This technique for controlling the details of an instruction execution introduces several important characteristics of the instruction set described below. The first is that there are many combinations of data type and instruction which are not valid. In addition, for those instructions which require two operands, the data type for the two data fields which are to be manipulated by the instruction must be similar. Finally, the use of descriptors results in a significant reduction in the total number of instructions in the instruction set, while at the same time the operations which can occur during the execution of each instruction can be quite complex because of the broad range of data types which are supported by the processor. This is reflected in the instruction descriptions below. Each of these paragraphs contains a general description of the instruction, a listing of the applicable data types, a description of the execution sequence for each data type, and a listing of the applicable hexidecimal Op Codes.

4.7.2 Format of the MCS65E4 Op Codes

All MCS65E4 instructions are contained within one byte. Bit 7 of each Op Code is reserved for use as a TRAP bit to cause an Instruction Access Trap. The remaining bits are used to specify one of sixteen instruction modes (bits 0-3) and one of the instructions within the selected mode. This format can be depicted as follows:



The instruction set can be organized into three major groups. The first group contains the basic data manipulation instructions (Add, subtract, etc.). The second contains all instructions which control program flow (branches, Jump to subroutine, etc.). Group 3 contains the advanced arithmetic and logic operations. Each of these groups is discussed separately below.

Figure 5.8 below summarizes the instruction set format for the

MCS65E4.

Format	Mnemonic
I,I,R	ADD, SUB, MUL, DIV AND, OR, EOR, MOD
I,IR	ADD, SUB, MUL, DIV AND, OR, EOR, MOD
I,R	ABS, NEG, INC, DEC SQRT, MOV, LEADZ, LEAD1
IR	ABS, NEG, INC, DEC SQRT, CLR, SET
I,I,# I,I,##	BEQ, BNE, BGT, BGE
I,# I,##	BEQZ, BNEZ, BPOS, BMI
I,I,R	CEQ, CNE, CGT, CGE
I,IR,S	FIND, DETC, NDET, DETR
(I,I,R),(I,IR)	SHM
I	CALL, JMP TASK, IOS, RTE
(IR),(I,I),# (IR),(I,I),##	BR, JSR, BDEC, BCOND
No Operand	RTS, RESET, SC, SCM
I,R	PTR, DTYP, CNVT
Special	EVAL

R: Result, I: Input, RI: Input and Result, S: String
( ) : Optional

4.7.3 Basic Arithmetic and Logic Operations

4.7.3.1 Introduction

The arithmetic and logic instructions can be organized into a set of operations which must be performed on two separate data fields and a set of operations which involve only a single data field. The first set of instructions will contain either two or three operands while one or two operands must be provided in the second set.

The following is a summary of the mnemonics, Op Codes and format for the basic arithmetic and logic instructions.

Group 1 Instructions (utilizing two data fields)

Instruction	Mnemonic	Two Operand Op Code	Three Operand Op code
Add	ADD		
Subtract	SUB		
Multiply	MUL		
Divide	DIV		
Modulus	MOD		
Logic And	AND		
Logic Or	OR		
Exclusive Or	EOR		

Format

Three operand: Op-Code OP1,OP2,OP3

Operand 1 (OP1) - Specifies the first source field
Operand 2 (OP2) - Specifies the second source field.
Operand 3 (OP3) - Specifies the result field.

Two operand: Op-Code OP1,OP2

Operand 1 (OP1) - Specifies the first source field.
Operand 2 (OP2) - Specifies both the second source field and the results field.

Group 2 Instructions (utilizing one data field)

Instruction	Mnemonic	Single Operand Op Code	Two Operand Op code
Absolute Value	ABS		
Negate	NEG		
Increment	INC		
Decrement	DEC		
Square Root	SQRT		
Move	MOV		
Convert	CNVT		
Clear	CLR		

Format

Two operand: Op-Code OP1,OP2

Operand 1 (OP1)- Specifies the only source field
Operand 2 (OP2)- Specifies the result field.

One operand: Op-Code OP1

Operand 1 (OP1)- Specifies both the source field
and the results field.

```
*********
*  ADD  *
*********
```

4.7.3.2 Add

Format: ADD OP1,OP2,OP3
         ADD OP1,OP2

Description:

Adds the contents of the data field specified by operand 2 to the contents of the data field specified by operand 1 and places the results in the data field specified by either operand 2 (two operand addressing) or operand 3 (three operand addressing).

Valid Data Types:

Binary (Byte, Integer and String)

Performs a binary add operation treating the byte and string data as a positive integer.

Real

Performs a floating point add operation. Input data is assumed to be normalized. After the add operation is complete, the results are normalized before being stored in the results field. Real operands cannot be mixed with Binary or BCD data fields.

BCD

Performs a signed, packed BCD add operation. All operands must be the same type and length.

Immediate Operands

Immediate operands can be specified in combination with any of the other data types. The immediate operand assumes the data type of the second input operand. If both input operands are Immediate data, the data type is assumed to be integer.

Op Codes:

Two operand addressing- ______
Three operand addressing-______

```
*********
*  SUB  *
*********
```

4.7.3.3 Subtract

Format: SUB OP1,OP2,OP3
         SUB OP1,OP2

Description:

CASE1 : Subtracts the contents of the data field specified by operand 1 from the contents of the data field specified by operand 2 and places the results in the data field specified by operand 3 (three operand addressing). Example : SUB a,b,c => c = b - a

CASE2 : Subtracts the contents of the data field specified by operand 1 from the contents of the data field specified by operand 2 and places the results in the data field specified by operand 2 (two operand addressing). Example : SUB a,b => b = b - a

Valid Data Types: Same as Add Instruction.

Op Codes:

Two operand addressing- ------
Three operand addressing- ------

```
*********
*  MUL  *
*********
```

4.7.3.4 Multiply

Format: MUL OP1,OP2,OP3
         MUL OP1,OP2

Description:

CASE1 : Multiplies the contents of the data field specified by operand 1 times the contents of the data field specified by operand 2 and places the results in the data field specified by operand 3 (three operand addressing).

CASE2 : Multiplies the contents of the data field specified by operand 1 times the contents of the data field specified by operand 2 and places the results in the data field specified by operand 1 (two operand addressing).

Valid Data Types:

	yes	no
Integer	x	
Bcd	x	
Real	x	
String	x	

Notes:

If the data field specified by OP1 and OP2 are string type, only the first eisth bytes will be used for multiplication. If the result field is string data type, the multiplication of the inputs (either string or integer) will be stored into the first sixteen bytes of the resultant string (assuming the result field is greater than or equal to 16 bytes).

Restrictions :

The length and type of operands 1 and 2 must be equal.

Op Codes:

Two operand addressing- ______
Three operand addressing- ______

```
*********
*  DIV  *
*********
```

4.7.3.5 Divide

Format: DIV OP1,OP2,OP2
         DIV OP1,OP2

Description:

CASE1 : Divide the contents of the data field specified by operand 2 by the contents of the data field specified by operand 1 and places the results into the data field specified by operand 3 (three operand addressing).

CASE2 : Divides the contents of the data field specified by operand 2 into the contents of the data field specified by operand 1 and places the results into the data field specified by operand 2 (two operand addressing).

Valid Data Types:

	yes	no
Integer	x	
Bcd	x	
Real	x	
String	x	

Restrictions:

The length and type of operands 1 and 2 must be equal.

Op Codes:

Two operand addressing- ______
Three operand addressing- ______

```
*********
*  AND  *
*********
```

4.7.3.6 And

Format: AND OP1,OP2,OP3
        AND OP1,OP2

Description:

CASE1 : Performs a logic AND operation between each bit of the data field specified by operand 1 and the corresponding bit of the data field specified by operand 2 and places the results in the data field specified by operand 3 (three operand addressing). The following diagram depicts the AND operation:

```
              OP2
             0   1
          |----|----|
        0 |  0 | 0  |
OP3 =     |----|----|   => OP1
        1 |  0 | 1  |
          |----|----|
```

CASE2 : Performs a logic AND operation between each bit of the data field specified by operand 1 and the corresponding bit of the data field specified by operand 2 and places the results in the data field specified by operand 2 (two operand addressing). The following diagram depicts the AND operation:

```
              OP2
             0   1
          |----|----|
        0 |  0 | 0  |
OP1 =     |----|----|   => OP1
        1 |  0 | 1  |
          |----|----|
```

Valid Data Types:

```
            yes  no
           |---|---|
Integer    | x |   |
           |---|---|
Bcd        | x |   |
           |---|---|
Real4      | x |   |
           |---|---|
Real8      | x |   |
           |---|---|
Real10     |   | x |
           |---|---|
```

String | x | | |
|---|---|

Restrictions:

The length and type of operands 1 and 2 must be equal.

Op Codes:

Two operand addressing- ------
Three operand addressing- ------

```
********
*  OR  *
********
```

4.7.3.7 Or

Format: OR OP1,OP2,OP3
OR OP1,OP2

Description:

CASE1 : Performs a logic OR operation between each bit of the data field specified by operand 1 and the corresponding bit of the data field specified by operand 2 and places the results in the data field specified by operand 3 (three operand addressing). The following diagram depicts the AND operation:

```
              OP2
             0   1
          |----|----|
        0 |  0 | 1  |
OP3 =     |----|----|  => OP1
        1 |  1 | 1  |
          |----|----|
```

CASE2 : Performs a logic OR operation between each bit of the data field specified by operand 1 and the corresponding bit of the data field specified by operand 2 and places the results in the data field specified by operand 2 (two operand addressing). The following diagram depicts the OR operation:

```
              OP2
             0   1
          |----|----|
        0 |  0 | 1  |
OP1 =     |----|----|  => OP1
        1 |  1 | 1  |
          |----|----|
```

Valid Data Types:

```
            yes  no
          |---|---|
Integer   | x |   |
          |---|---|
Bcd       | x |   |
          |---|---|
Real4     | x |   |
          |---|---|
Real8     | x |   |
          |---|---|
Real10    |   | x |
```

String	x	

Restrictions:

The length and type of operands 1 and 2 must be equal.

Op Codes:

Two operand addressing- ______
Three operand addressing- ______

```
*********
*  EOR  *
*********
```

4.7.3.8 Exclusive Or

Format: EOR OP1,OP2,OP3
        EOR OP1,OP2

Description:

CASE1 : Performs a logic EOR operation between each bit of the data field specified by operand 1 and the corresponding bit of the data field specified by operand 2 and places the results in the data field specified by operand 3 (three operand addressing). The following diagram depicts the EOR operation:

```
          OP2
          0   1
        |---|---|
      0 | 0 | 1 |
OP3 =   |---|---|    => OP1
      1 | 1 | 0 |
        |---|---|
```

CASE2 : Performs a logic EOR operation between each bit of the data field specified by operand 1 and the corresponding bit of the data field specified by operand 2 and places the results in the data field specified by operand 2 (two operand addressing). The following diagram depicts the EOR operation:

```
          OP2
          0   1
        |---|---|
      0 | 0 | 1 |
OP3 =   |---|---|    => OP1
      1 | 1 | 0 |
        |---|---|
```

Valid Data Types:

```
          yes  no
         |---|---|
Integer  | x |   |
         |---|---|
Bcd      | x |   |
         |---|---|
Real4    | x |   |
         |---|---|
Real8    | x |   |
         |---|---|
```

Real10		x
Strins	x	

Restrictions:

The length and type of operands 1 and 2 must be equal.

Op Codes:

Two operand addressing- _____
Three operand addressing- _____

```
*********
*  MOD  *
*********
```

4.7.3.9 Modulus (Remainder Function)

Format: MOD OP1,OP2,OP3
        MOD OP1,OP2

Description:

CASE1 : Divide the contents of the data field specified by operand 2 by the contents of the data field specified by operand 1 and places the remainder into the data field specified by operand 3 (three operand addressing).

CASE2 : Divides the contents of the data field specified by operand 2 by the contents of the data field specified by operand 1 and places the remainder into the data field specified by operand 2 (two operand addressing).

Valid Data Types:

	yes	no
Integer	x	
Bcd	x	
Real		x
String	x	

Restrictions:

The length and type of operands 1 and 2 must be equal.

Valid Data Types:

Op Codes:

Two operand addressing- ______
Three operand addressing- ______

```
*********
*  ABS  *
*********
```

4.7.3.10 Absolute Value

Format: ABS OP1,OP2
        ABS OP1

Description:

CASE1 : Changes the data field specified by operand 1 into a positive number storing the results into the data field specified by operand 2 (two operand addressing).

CASE2 : Changes the the data field specified by operand 1 into a positive number storing the results into the same data field (single operand addtessing).

Valid Data Types:

	yes	no
Integer	x	
Bcd	x	
Real	x	
String		x

Notes: For integer data type, changing a negative number into a positive number will involve more than changing the sign bit since integer data is stored in 2's complement form.

Op Codes:

One operand addressing- ______
Two operand addressing- _____

```
*********
*  NEG  *
*********
```

4.7.3.11 Negate Value

Format: NEG OP1,OP2
        NEG OP1

Description:

CASE1 : Inverts the sign of the data field specified by operand 1 storing the results into the data field specified by operand 2 (two operand addressing).

CASE2 : Inverts the sign of the data field specified by operand 1 storing the results into the same data field (single operand addtessing).

Valid Data Types:

	yes	no
Integer	x	
Bcd	x	
Real	x	
String		x

Notes:

For integer type, the negation will be more than just change the sign (i.e. twos compliment arithmetic).

Op Codes:

One operand addressing- ------
Two operand addressing- -----

```
*********
*  INC  *
*********
```

4.7.3.12 Increment

Format: INC OP1,OP2
        INC OP1

Description:

CASE1 : Adds one to the data specified by operand 1, storing the results back into the data field specified by operand 2 (two operand addressing).

CASE2 : Adds one to the data specified by the operand storing the results back into the same data field (single operand addressing).

Valid Data Types:

	yes	no
Integer	x	
Bcd	x	
Real		x
String		x

Op Codes:

One operand addressing- ------
Two operand addressing- -----

```
*********
*  DEC  *
*********
```

4.7.3.13 Decrement

Format: DEC OP1,OP2
        DEC OP1

Description:

CASE1 : Subtracts one from the data specified by operand 1, storing the results into the data field specified by operand 2 (two operand addressing).

CASE2 : Subtracts one from the data specified by the operand, storing the results back into the same data field (single operand addressing).

Valid Data Types:

	yes	no
Integer	x	
Bcd	x	
Real		x
String		x

Op Codes:

One operand addressing- ------
Two operand addressing- -----

****
* SQRT *
****

4.7.3.14 Square Root

Format: SQRT OP1,OP2
SQRT OP1

Description:

CASE1 : Determines the square root of the data specified by operand 1, storing the results into the data field specified by operand 2 (two operand addressing).

CASE2 : Determines the square root of the data specified by the operand, storing the results into the same data field (single operand addressing).

Valid Data Types:

	yes	no
Integer	x	
Bcd	x	
Real	x	
String		x

Op Codes:

One operand addressing- ------
Two operand addressing- -----

```
*********
*  MOV  *
*********
```

4.7.3.15 Move

Format: MOV OP1,OP2

Description:

Transfers the contents of the data field specified by operand 1 into the data field specified by operand 2.

Valid Data Types:

	yes	no
Integer	x	
Bcd	x	
Real	x	
String	x	

Restrictions:

The length and type of operands 1 and 2 must be equal.

Op Code- ------

****
* LEADZ *
****

4.7.3.16 Leadz

Format: LEADZ OP1,OP2

Description:

Searches the data field referenced by operand 1 for the first occurance of a zero bit. The results are returned in the data field specified by operand 2.

Valid Data Types for OP1:

	yes	no
Integer4	x	
Integer8	x	
Bcd	x	
Real	x	
String	x	

Restrictions:

The data field referenced by OP2 must be an integer.

For ten byte real only the mantissa will be searched.

Example:

If the first zero is in the fifth position of the fourth byte then a value of 30 ((4-1)*8 + 4)) will be returned. If the first zero bit is the first position then a value of 0 will be returned.

Op Code- ------

****
* LEAD1 *
****

4.7.3.17 Lead1

Format: LEAD1 OP1,OP2

Description:

Searches the data field referenced by operand 1 for the first occurance of a one bit. The results are returned in the data field specified by operand 2.

Valid Data Types for OP1:

	yes	no
Integer4	x	
Integer8	x	
Bcd	x	
Real	x	
String	x	

Restrictions:

The data field referenced by Operand 2 must be an integer.

For ten byte real only the mantissa will be searched.

Example:

If the first one is in the second position of the third byte then a value of 17 ((3-1)*8 + 1)) will be returned.

Note: If the first one bit is the first position then a value of 0 will be returned.

Op Code- ------

```
*********
*  CLR  *
*********
```

4.7.3.18 Clear

Format: CLR OP1

Description:

Loads an arithmetic zero into the data field specified by the single operand contained in the instruction.

Valid Data Types:

	yes	no
Integer	x	
Bcd	x	
Real	x	
String	x	

Op Code- ------

```
*********
*  SET  *
*********
```

4.7.3.19 Set

Format: SET OP1

Description:

Set on all bits (i.e makes bits logic 1) in the data field specified by the single operand contained in the instruction.

Valid Data Types:

	yes	no
Integer	x	
Bcd	x	
Real	x	
String	x	

Op Code- ------

4.7.5 Program Control Instructions

4.7.5.1 Introduction

The set of program control instructions described in this section consists of those operations which directly affect program flow during process execution. This includes conditional branching, unconditional branch and jump instructions, subroutine call and return, system call and return, and miscellaneous process control intructions. These instructions will be organized into four groups as follows:

1. Compare and branch operations.
2. Test and branch operations.
3. Single operand control functions.
4. Miscellaneous (no operand) control functions.

```
*********
*  BEQ  *
*********
```

4.7.5.2 Compare and Branch If Equal

Format: BEQ OP1,OP2,#
         BEQ OP1,OP2,##

Description:

Compare the data field referenced by OP1 with the data field referenced by OP2 and branch conditionally (#: one byte offset; ##: two byte offset) if they are equal.

Notes:

Location # is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -128 and less than or equal to +127 bytes from the start of the next instructions opcode.

Location ## is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -65536 and less than or equal to +65535 bytes from the start of the next instructions opcode.

Restrictions:

The data fields referenced by OP1 and OP2 must be the same type and length.

Op Codes:

```
*********
*  BNE  *
*********
```

4.7.5.3 Compare and Branch If Not Equal

Format:
```
BNE   OP1,OP2,*
BNE   OP1,OP2,**
```

Description:

Compare the data field referenced by OP1 with the data field referenced by OP2 and branch conditionally (* one byte offset; ** two byte offset) if they are not equal.

Notes:

Location * is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -128 and less than or equal to +127 bytes from the start of the next instructions opcode.

Location ** is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -65536 and less than or equal to +65535 bytes from the start of the next instructions opcode.

Restrictions:

The data fields referenced by OP1 and OP2 must be the same type and length.

Op Codes:

```
*********
*  BGT  *
*********
```

4.7.5.4 Compare and Branch If Greater Than

Format:
```
BGT   OP1,OP2,#
BGT   OP1,OP2,##
```

Description:

Branch conditionally (# one byte offset; location ## two byte offset) if the data field referenced by OP1 is greater than the data field referenced by OP2.

Notes:

Location # is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -128 and less than or equal to +127 bytes from the start of the next instructions opcode.

Location ## is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -65536 and less than or equal to +65535 bytes from the start of the next instructions opcode.

Restrictions:

The data fields referenced by OP1 and OP2 must be the same type and length.

Op Codes:

```
*********
*  BGE  *
*********
```

4.7.5.5 Compare and Branch If Greater Than

Format: BGE OP1,OP2,#
         BGE OP1,OP2,##

Description:

Branch conditionally (# one byte offset; location ## two byte offset) if the data field referenced by OP1 is greater than or equal to the data field referenced by OP2.

Notes:

Location # is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -128 and less than or equal to +127 bytes from the start of the next instructions opcode.

Location ## is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -65536 and less than or equal to +65535 bytes from the start of the next instructions opcode.

Restrictions:

The data fields referenced by OP1 and OP2 must be the same type and length.

Op Codes:

```
**********
*  BEQZ  *
**********
```

4.7.5.6 Branch if Equal

Format: BEQZ OP1,#
         BEQZ OP1,##

Description:

Branch conditionall (# one byte offset; location ## two byte offset) if data field referenced by OP1 is equal to zero.

Notes:

Location # is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -128, and less than or equal to +127 bytes from the start of the next instructions opcode.

Location ## is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -65536 and less than or equal to +65535 bytes from the start of the next instructions opcode.

Op Codes:

```
**********
*  BNEZ  *
**********
```

4.7.5.7 Branch If Not Equal

Format:
```
BNEZ  OP1,#
BNEZ  OP1,##
```

Description:

Branch conditionally (# one byte offset; location ## two byte offset) if data field referenced by OP1 is not equal to zero.

Notes:

Location # is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -128 and less than or equal to +127 bytes from the start of the next instructions opcode.

Location ## is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -65536 and less than or equal to +65535 bytes from the start of the next instructions opcode.

Op Codes:

****
* BPOS *
****

4.7.5.8 Branch If Positive

Format: BPOS OP1,#
BPOS OP1,##

Description:

Compare the data field referenced by OP1 and branch conditionally (# one byte offset; ## two byte offset) if the most significant bit is set to logic 0.

Notes:

Location # is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -128 and less than or equal to +127 bytes from the start of the next instructions opcode.

Location ## is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -65536 and less than or equal to +65535 bytes from the start of the next instructions opcode.

For bcd, integer and real, the sign bit which is always in the most significant bit, will be tested for logic 0. If the value is positive a branch will occur. The BMI instruction (see Section 4.7.5.9) can be used for branch on negative numbers.

Op Codes:

```
*********
*  BMI  *
*********
```

4.7.5.9 Branch If Minus

Format: BMI OP1,#
BMI OP1,##

Description:

Compare the data field referenced by OP1 and branch conditionally (# one byte offset; location ## two byte offset) if the most significant bit is set to logic 1.

Notes:

Location # is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -128 and less than or equal to +127 bytes from the start of the next instructions opcode.

Location ## is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -65536 and less than or equal to +65535 bytes from the start of the next instructions opcode.

For bcd, integer and real, the sign bit which is always in the most significant bit, will be tested for logic 1. If the value is negative a branch will occur.

Op Codes:

```
********
*  BR  *
********
```

4.7.5.10 Branch Relative Unconditionally

Format: BR *
        BR **

Description:

Branch (* one byte offset; location ** two byte offset) unconditionally.

Notes:

Location * is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -128 and less than or equal to +127 bytes from the start of the next instructions opcode.

Location ** is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -65536 and less than or equal to +65535 bytes from the start of the next instructions opcode.

Locations * and ** can not specify a variable which could be used for indirect addressing. Instead, the generated address is a positive or negative offset from the opcode of the next instruction. Indirect addressing can be accomplished with the JMP instruction (see Section 4.7.5.11).

```
*********
*  JMP  *
*********
```

4.7.5.11 Jump Logical Address Unconditionally

Format: JMP OP1

Description:

Jump unconditionally to the address contained in the data field referenced by OP1.

Notes:

If the destination location is fixed (i.e. not calculated at run time but rather assemble time) the data field referenced by OP1 should be an immediate value. However, if the destination address is not known at assemble time then OP1 should be a variable. In fact, OP1 can define an array structure which could be used to establish an array of addresses. Thus, a 'case' type of instruction can be formed (see example below).

Example

```
        .
        .
   JMP TABLE[I]      OP1 = References an array structure.
        .
        .
LABLA
        .
        .
        .
LABLB
        .
        .
        .
LABLC
        .
        .
```

```
TABLE =   |--------------------|
          |  Descriptor Header |
          |--------------------|
          |     LABLA          |
          |--------------------|
          |     LABLB          |
          |--------------------|
          |     LABLC          |
          |--------------------|
```

The processor does the following to obtain the JMP address:

Get TABLE's operand control byte which indicates primary

based descriptor accessing and a two byte offset. The contents of the primary register plus the two byte offset point to the descriptor header.

Determine that TABLE is a simple array of attached ordinals (TABLE's descriptor header indicates this).

Fetch subsript operand (variable I) for the index into the array.

Determine the address of TABLE[I] (the address of the raw data is TABLE + 3*I (see SECTION ______) for traversing through an array).

The data field (three byte ordinal) referenced by TABLE[I] is the logical address for the program counter.

The program continues with the appropiate program counter.

In the above example if I equals 0 then the program will jump to LABLA. If I equals 1 then the program will jump to LABLB and if I equals 2 then the program will jump to LABLC.

```
*********
*  BSR  *
*********
```

4.7.5.12 Branch to Subroutine

Format: BSR #
        BSR ##

Description:

Branch to subroutine at location # (one byte offset) or location ## (two byte offset) unconditionally.

Notes:

Location # is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -128 and less than or equal to +127 bytes from the start of the next instructions opcode.

Location ## is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -65536 and less than or equal to +65535 bytes from the start of the next instructions opcode.

Locations # and ## can not specify a variable which could be used for indirect addressing. Instead, the generated address is a positive or negative offset from the opcode of the next instruction. Indirect addressing can be accomplished with the JSR instruction (see Section 4.7.5.13).

The return address from the subroutine is saved onto the stack as described in Section_____. Exiting from the subroutine is accomplished by executing the return from subroutine (RTS) instruction (see Section 4.7.5.18).

```
*********
*  JSR  *
*********
```

4.7.5.13 Jump To Subroutine

Format: JSR OP1

Description:

Branch to subroutine at the address contained in the data field referenced by OP1.

Notes:

If the destination location is fixed (i.e. not calculated at run time but rather assemble time) the data field referenced by OP1 should be an immediate value. However, if the destination address is not known at assemble time then OP1 should be a variable. In fact, OP1 can define an array structure which could be used to establish an array of addresses (see example below).

The return address from the subroutine is saved onto the stack as described in Section_____. Exiting from the subroutine is accomplished by executing the return from subroutine (RTS) instruction (see Section 4.7.5.18).

Example

```
        .
        .
   JSR TABLE[I]      OP1 = References an array structure.
        .
        .
LABLA
        .
        .
        .
LABLB
        .
        .
        .
LABLC
        .
        .
        .


TABLE =   |--------------------|
          |  Descriptor Header |
          |--------------------|
          |     LABLA          |
          |--------------------|
          |     LABLB          |
          |--------------------|
          |     LABLC          |
```

|--------------------|

The processor does the following to obtain the subroutine address:

Get TABLE's operand control byte which indicates primary based descriptor accessing and a two byte offset. The contents of the primary register plus the two byte offset point to the descriptor header.

Determine that TABLE is an array of attached ordinals (TABLE's descriptor header indicates this).

Fetch subsript operand (variable I) for the index into the array.

Determine the address of TABLE[I] (the address of the raw data is TABLE + 3*I (see SECTION ______) for traversing throush an array).

The data field (three byte ordinal) referenced by TABLE[I] is the logical address for the program counter.

The program continues with the appropiate program counter.

In the above example if I equals 0 then the program will jump to LABLA. If I equals 1 then the program will jump to LABLB and if I equals 2 then the program will jump to LABLC.

****
* BDEC *
****

4.7.5.14 Decrement and Branch

Format: BDEC OP1,#
BDEC OP1,##

Description:

Decrement the data field referenced by OP1 and branch conditionally (# one byte offset; location ## two byte offset) if it is not equal to 0.

Notes:

Location # is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -128 and less than or equal to +127 bytes from the start of the next instructions opcode.

Location ## is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -65536 and less than or equal to +65535 bytes from the start of the next instructions opcode.

This instruction is very useful in a loop situation. OP1 first contains a positive integer which serves as a counter. The MCS65E4 will execute the loop until this counter reaches zero and will then proceed to the next instruction.

Op Codes:

```
**********
*  BCOND  *
**********
```

4.7.5.15 Branch On Condition

Format: BCOND OP1,OP2,OP3

Description:

Add the data field referenced by OP1 to the data field referenced by OP2 storing the results back into the field referenced by OP2. The result is compared to the field referenced by OP3. The processor will then conditionally branch to location # (one byte offset) or location ## (two byte offset).

Notes:

The BCOND instruction is similar to the FORTRAN do loop. The field referenced by OP1 is the 'stepping value', the field referenced by OP3 is the limit value, and the field referenced by OP2 is the current value within the loop.

The 'stepping value' can either be positive or negative. If it is positive the current value is incremented until it is greater than or equal to the limit value. If the 'stepping value' is negative the current value is decremented until it is less than or equal to the limit value.

Relative Value # is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -128 and less than or equal to +127 bytes from the start of the next instructions opcode.

Relative Value ## is determined by subtracting the program counter from the branch destination. This offset must be greater than or equal to -65536 and less than or equal to +65535 bytes from the start of the next instructions opcode.

Locations # and ## can not specify a variable which could be used for indirect addressing. Instead, the generated address is a positive or negative offset from the opcode of the next instruction.

```
********
*  SC  *
********
```

4.7.5.16 System Call

Format: SC

Description:

Causes the processor to execute a trap through the exception vector located at the address __________ on the limit page. This operation is described in more detail in Section ----------.

Restrictions:

Must be serviced within operating system process (U = 0).

```
*********
*  SCM  *
*********
```

4.7.5.17 System Call Message

Format: SCM #

Description:

Causes the processor to execute a trap through the exception vector located at address ______ within the limit page. At the same time, the 8 bit message (#) which follows the operand is passed to the operating system. This operation is described in more detail in Section _________.

Restrictions:

Must be serviced within operating system process.

```
*********
*  RTS  *
*********
```

4.7.5.18 Return From Subroutine

Format: RTS

Description:

Return from subroutine.

Notes:

The JSR or BSR was used to call the subroutine and subsequently the return address was saved on the process stack. The RTS instruction is used to get back to the caller.

```
*********
*  RTE  *
*********
```

4.7.5.19 Retrurn From Exception

Format: RTE OP1

Description:

Return to the middle of instruction execution. The data field referenced by OP1 contains all the necessary return information.

Notes:

This instruction should only be executed when returning to an instruction within the same process. The TASK or IOS instruction should be used when returning to an instruction within another process.

```
*********
*  IOS  *
*********
```

4.7.5.20 Initiate Operating System

Format: IOS OP1

Description:

Initiates a higher - level operating system process. The user/supervisor flag remains logic 0. The kernal flag is set to a logic 0. The operation is described in detail in Section ________.

Notes:

The data field referenced by OP1 specifies the logical address of the new process.

Restrictions:

Must be serviced within an Operating System or Kernel process.

****
* TASK *
****

4.7.5.21 TASK

Format: TASK OP1

Description:

Initiates a user process. Both the user/supervisor flag and the kernal flag are set to 0. This operation is described in detail in Section ________.

Notes:

The data field referenced by OP1 specifies the logical address of the process parameter list (PPL) for the new process.

Restrictions:

Must be serviced within an operating system process.

4.7.6 Advanced Operations

4.7.6.1 Introduction

This group of instructions contains a number of powerful instructions which greatly facilitate the control of program sequencing within the MCS65E4 software. These instructions allow the processor to compare tw data fields and to set a Boolean variable as a function of the data in the two fields. In addition, a full set of string instructions, data shifting instructions and data conversion instructions is provided along with a num of instructions which are specifically designed to facilitate the control of the data within the MCS65E4 process. Each of these instructions is described below.

****
* RESET *
****

4.7.6.2 Reset

Format: RESET

Description:

Causes the IORES bit in the bus status word to so low for 16 cycles.

Notes:

The instruction has no operands.

```
*********
*  CEQ  *
*********
```

4.7.6.3 Compare Two Operands For Equality

Format: CEQ OP1,OP2,OP3

Description:

Compare for equality the data field referenced by OP1 with the data field referenced by OP2 and place the boolean result into the low order bit position of the field referenced by OP3.

Notes:

If the field referenced by OP1 is equal to the field referenced by OP2 then the value 0000 0001 will be put into the data field referenced by OP3. If the fields are not equal the value 0000 0000 will be put into the field referenced by OP3.

The data field referenced by OP3 should be byte data type. If it is not, only the least sisnificant byte will be affected by the instruction.

Restrictions:

The fields referenced by OP1 and OP2 must be the same type and lensth.

Example:

Before CEQ instruction:
OP1 = References a data field whose value is 16
OP2 = References a data field whose value is 16
OP3 = References a data field whose value is 0111 0001

After CEQ instruction:
OP1 = References a data field whose value is 16
OP2 = References a data field whose value is 16
OP3 = References a data field whose value is 0000 0001

```
*********
*  CNE  *
*********
```

4.7.6.4 Compare Two Operands For Inequality

Format: CNE OP1,OP2,OP3

Description :

Compare for inequality, the data field referenced by OP1 with the data field referenced by OP2 and put the boolean result into the low order bit position of the field referenced by OP3.

Notes:

If the field referenced by OP1 is not equal to the field referenced by OP2 then the value 0000 0001 will be put into the data field referenced by OP3. If the fields are equal the value 0000 0000 will be put into the field referenced by OP3.

The data field referenced by OP3 should be byte data type. If it is not, only the least significant byte will be effected.

Restrictions:

The fields referenced by OP1 and OP2 must be the same type and length.

Example:

Before CNE instruction:
OP1 = References a data field whose value is 13
OP2 = References a data field whose value is 16
OP3 = References a data field whose value is 0111 0001

After CNE instruction:
OP1 = References a data field whose value is 13
OP2 = References a data field whose value is 16
OP3 = References a data field whose value is 0000 0001

```
*********
*  CGT  *
*********
```

4.7.6.5 Compare Two Operands For Greater Than

Format: CGT OP1,OP2,OP3

Description:

Compare the data field referenced by OP1 with the data field referenced by OP2. If the data field referenced by OP1 is greater than the data field referenced by OP2 then put the boolean result into the low order bit position into the field referenced by OP3.

Notes:

If the field referenced by OP1 is greater than the field referenced by OP2 then the value 0000 0001 will be put into the data field referenced by OP3 else the value 0000 0000 will be used.

The data field referenced by OP3 should be byte data type. If it is not, only the least significant byte will be affected.

Restrictions:

The fields referenced by OP1 and OP2 must be the same type and length.

Example:

Before CGT instruction:
OP1 = References a data field whose value is 47
OP2 = References a data field whose value is 16
OP3 = References a data field whose value is 0111 0001

After CGT instruction
OP1 = References a data field whose value is 47
OP2 = References a data field whose value is 16
OP3 = References a data field whose value is 1110 0011

```
*********
*  CGE  *
*********
```

4.7.6.6 Compare Two Operands For Greater Than Or Equal

Format: CGE OP1,OP2,OP3

Description:

Compare the data field referenced by OP1 with the data field referenced by OP2. If the data field referenced by OP1 is greater than or equal to the data field referenced by OP2 then put the boolean result into the low order bit position into the field referenced by OP3.

Notes:

If the field referenced by OP1 is greater than or equal to the field referenced by OP2 then the value 0000 0001 will be put into the data field referenced by OP3 else the value 0000 0000 will be used.

The data field referenced by OP3 should be byte data type. If it is not, only the least significant byte will be affected.

Restrictions:

The fields referenced by OP1 and OP2 must be the same type and length.

Example:

Before CGE instruction:
OP1 = References a data field whose value is 47
OP2 = References a data field whose value is 16
OP3 = References a data field whose value is 0111 0001

After CGE instruction:
OP1 = References a data field whose value is 47
OP2 = References a data field whose value is 16
OP3 = References a data field whose value is 0000 0001

****
* FIND *
****

4.7.6.7 Find String

Format : FIND OP1,OP2,OP3

Description

Find the first occurance of a specified data item within a given string.

OP1:

The data field referenced by OP1 contains the search argument and must be of type byte, integer, ordinal or string.

OP2:

The data field referenced by OP2 contains the starting point of the string data field referenced by OP3.

OP3:

The data field referenced by OP3 represents the string field to be searched. The starting point within the data field referenced by OP3 is contained in the data field referenced by OP2.

Notes:

The data referenced by Operand 3 must be of type string. The length of this string is defined by its descriptor.

If the data referenced by Operand 1 is of type string, then only the first 8 bytes are used for the search argument.

It is the users responsibility to initialize the data field referenced by OP2 with the initial starting point within the string. The search will begin from the first byte of the string if the data field referenced by OP2 contains -1. If a match is detected, the data field referenced by OP2 will contain a positive number which will indicate the byte position within the string. When the search is complete, the data field referenced by OP2 will contain -2.

Example:

Before FIND instruction:

OP1 = References a data field whose value is: NOW BR
OP2 = References a data field whose value is: -1
OP3 = References a data field whose value is: HOW NOW BROWN COW.

After FIND instruction:

OP1 = References a data field whose value is: NOW BR
OP2 = References a data field whose value is: +9
OP3 = References a data field whose value is: HOW NOW BROWN COW.

```
**********
*  DETC  *
**********
```

4.7.6.8 Detect Character in String

Format: DETC OP1,OP2,OP3

Description:

Find the first byte in a string which matches a byte from a given set of bytes.

OP1:

The data field referenced by OP1 contains the set of bytes which will be matched against the search string.

OP2:

The data field referenced by OP2 contains the starting point of the string data field referenced by OP1.

OP3:

The data field referenced by OP3 represents the string field to be searched. The starting point within the data field referenced by OP3 is contained in the data field referenced by OP2.

Notes:

The data referenced by Operand 3 must be of type string. The length of this string is defined by its descriptor.

If the data referenced by Operand 1 is of type string, then only the first 8 bytes are used for the search argument.

It is the users responsibility to initialize the data field referenced by OP2 with the initial starting point within the string. The search will begin from the first byte of the string if the data field referenced by OP2 contains -1.

The search stops if any byte from the set of bytes matches a byte within the search string. In that case, the data field referenced by OP2 will contain a positive number which will indicate the byte position within the string. When the search is complete, the data field referenced by OP2 will contain -2.

Example:

Before DETC instruction:

OP1 = References a data field whose value is: XYZW
OP2 = References a data field whose value is: -1

OP3 = References a data field whose value is: HOW NOW BROWN COW.

After DETC instruction:

OP1 = References a data field whose value is: XYZW
OP2 = References a data field whose value is: +2
OP3 = References a data field whose value is: HOW NOW BROWN COW.

****
* NDET *
****

4.7.6.9 Detect Character not in String

Format: NDET OP1,OP2,OP3

Description:

Find the first byte in a string which does not matches a byte from a given set of bytes.

OP1:

The data field referenced by OP1 contains the set of bytes which will be matched against the search string.

OP2:

The data field referenced by OP2 contains the starting point of the string data field referenced by OP3.

OP3:

The data field referenced by OP3 represents the string field to be searched. The starting point within the data field referenced by OP3 is contained in the data field referenced by OP2.

Notes:

The data referenced by Operand 3 must be of type string. The length of this string is defined by its descriptor.

If the data referenced by Operand 1 is of type string, then only the first 8 bytes are used for the search argument.

It is the users responsibility to initialize the data field referenced by OP2 with the initial starting point within the string. The search will begin from the first byte of the string if the data field referenced by OP2 contains -1.

The search stops if any byte from the set of bytes does not match a byte within the search string. In that case, the data field referenced by OP2 will contain a positive number which will indicate the byte position within the string. When the search is complete, the data field referenced by OP2 will contain -2.

Example:

Before NDET instruction:

OP2 = References a data field whose value is: ABCMCS
OP1 = References a data field whose value is: -1

OP3 = References a data field whose value is: MCS65E4

After NDET instruction:

OP1 = References a data field whose value is: ABCMCS
OP2 = References a data field whose value is: +3
OP3 = References a data field whose value is: MCS65E4

```
**********
*  DETR  *
**********
```

4.7.6.10 DETR

Format: DETR OP1,OP2,OP3

Description:

Search string for a single byte which is within a defined range.

OP1:

The data field referenced by OP1 contains the two byte range field which defines the lower and upper bound. The least significant byte of this field defines the upper bound and next consecutive byte defines the lower bound.

OP2:

The data field referenced by OP2 contains the starting point of the string data field referenced by OP3.

OP3:

The data field referenced by OP3 represents the string field to be searched. The starting point within the data field referenced by .OP3 is contained in the data field referenced by OP2.

Notes:

As stated above, the least significant byte of the field referenced by OP1 defines the lower bound and the next consecutive byte defines the upper bound of the range. During the search operation, any byte from the data field which is referenced by OP3 which is greater than or equal to the lower bound and less than or equal to the upper bound will halt the search operation.

It is the users responsibility to initialize the data field referenced by OP2 with the initial starting point within the string. The search will begin from the first byte of the string if the data field referenced by OP2 contains -1. When a byte is within the defined range, a positive number will be returned which indicates the byte position within the string. When the search is complete, the data field referenced by OP2 will contain -2.

Example:

Before DETR instruction:

OP1 = References a data field whose value is: AB12 B4D7

OP2 = References a data field whose value is: -1
OP3 = References a data field whose value is: 22BF 4567 891F

After DETR instruction:

OP1 = References a data field whose value is: AB12 B4D7
OP2 = References a data field whose value is: +1
OP3 = References a data field whose value is: 22BF 4567 891F

In this example the lower bound = D7 and the upper bound = B4. Since BF is greater than or equal to D7 and less than or equal to B4, the data field referenced by OP3 will contain +1.

```
*********
*  SHM  *
*********
```

4.7.6.11 Shift Multiple

Format:
```
SHM   OP1,OP2,OP3
SHM   OP1,OP2
```

Description:

CASE1 : Shift the data field referenced by OP1 by the number of bit positions specified by the data field referenced by OP2 storing the results into the data field referenced by OP3 (three operand addressing).

CASE2 : Shift the data field referenced by OP1 by the number of bit positions specified by the data field referenced by OP2 storing the results back into the field referenced by OP2 (two operand addressing).

Notes:

If the field referenced by OP2 is positive a left shift will occur. A negative number will cause a right shift.

The lenth of the data field to be shifted is specified in the descriptor of the field referenced by OP1 or in the operand control byte in the case of byte, integer or ordinal.

Logic 0 will fill the bit positions created by shift left or shift right unless the data field referenced by OP1 is an integer. In this case, a right shift operation will sign extend (i.e. the most significant bit will fill the bit positions).

For ten byte real, only the mantissa will be shifted.

Example:

```
.
.
.
SHM  OP1,OP2,OP3
.
.
.
```

Before SHM instruction:
OP1 = References a one byte field whose value is 1011 0001
OP2 = References a data field whose value is -2
OP3 = References a data field whose value is 1111 1111

After SHM instruction:
OP1 = References a one byte field whose value is 1011 0001

OP2 = References a data field whose value is -2
OP3 = References a data field whose value is 1110 1100

```
*********
*  PTR  *
*********
```

4.7.6.14 Point to Data Field

Format: PTR OP1,OP2

Description:

Returns the Logical Address of a Data Field.

OP1:

The PTR instruction will determine the logical address within the process of the field referenced by OP1. This field can be any data type but it can not be an immediate value or an internal register.

OP2:

The 24 bit result will be placed in the field referenced by OP2 which must be of type ordinal.

Notes:

The results of this operation will return the logical address of either string or scalar (e.g. byte, integer, real, etc.) type data. If the data field referenced by OP1 is not string or scalar type data (i.e. record or array type) the processor will search through the data structure until it encounters a basic data element from the structure.

This instruction is very useful to speed up the operation in the loop instead of having to calculate the address of the operand everytime. All that is needed is first calculate the address of the data item then update the pointer address.

```
Example:  PTR   TABLE[I],B
            .
            .
            .

TABLE = 12345678    (0th element - logical address = 001C00)
          .
          .
          .
        23456789    (8th element - logical adreess = 001C20)
```

Before PTR instruction:
OP1 = References the data field whose value is 23456789 with logical address 001C20.
B   = References the data field whose value is 000000

After PTR instruction:
OP1 = References the data field whose value is 23456789 with logical address 001C20.
B = References the data field whose value is 001C20

****
* DTYPE *
****

4.7.6.15 Get Data type

Format: DTYPE OP1,OP2

Description:

Determines the Data Type of a Data Field.

OP1:

The DTYPE instruction will determine the data type of the field referenced by OP1. This field can be any data type but it can not be an immediate value or an internal register.

OP2:

The results will be placed in the first byte of the field referenced by OP2 according to the table listed below (Result Table). If the resultant data type is string the next two consecutive bytes of the field referenced by OP2 will contain the string length.

Notes:

The results of this operation must return either string data type or scalar (e.g. byte, integer, real, etc.) data type. If the data field referenced by OP1 is not string or scalar data type (i.e. record or array type) the processor will search through the data structure until it encounters a basic data element from the structure.

The data field referenced by OP2 (result field) will contain a value which will reflect the descriptor header field. These values are shown below in the Result Table:

Field Referenced By OP1	Field Referenced By OP2
Byte	04
Ordinal	08
Two Byte Integer	0C
Four Byte Integer	10
Eight Byte Integer	14
Four Byte Real	18
Eight Byte Real	1C
Ten Byte Real	20
Four Byte BCD	24
Eight Byte BCD	28
Ten Byte BCD	2C
String	30

Example:

Before DTYPE instruction

OP1 = References the 4th element of an array of two byte integer.
OP2 = References the data field whose value is 00.

After DTYPE instruction

OP1 = References the 4th element of an array of two byte integer.
OP2 = References the data field whose value is 0C.

****
* CNVRT *
****

4.7.6.16 CNVRT

Description:

Transfers the contents of the data field specified by operand 1 into the data field specified by operand 2, converting the format of the data to that of the second data field in the process.

Valid Data Types:

Op Code- ------

****
* EVAL *
****

4.7.6.17 EVAL

The EVAL instruction allows the MCS65E4 to directly evaluate an arithmetic expression. This is accomplished by organizing the expression into Reverse Polish notation and attaching it to the EVAL Op Code.

The procedures for incorporating an expression into an EVAL instruction are as follows:

1. All of the data accessing procedures operate in the normal manner except that the data fields must be contained in memory, i. e., data contained in the internal registers cannot be referenced within the arithmetic expression.

2. Those operand control byte codes which would normally reference the internal registers (see Section ____) are replaced by the arithmetic and logic instructions described above. This specifies the operations to be performed as follows:

3. The final operation which must be performed in the expression is a MOVE operation. This must place the results into the desired results field.

Code	Operation
50	ADD
51	SUBTRACT
52	MULTIPLY
53	DIVIDE
54	AND
55	OR
56	EOR
57	MOD
58	ABS
59	NEG
5A	INC
5B	DEC
	CEQ
	CNE
	CGT
	CGE
5D	MOV
5E	LEADZ
5F	LEAD1

The operation of this instruction is illustrated in the example below.

Example:

The expression:

Y=((((A + (B+1) * C) / (D - E) AND (NEG F + (G OR H)))

can be converted to Reverse Polish notation as follows:

A B INC + C * D E - / F NEG G H OR + AND MOV

This expression can be converted to a single EVAL instruction as follows:

Field #	Contents	Comments
1		EVAL Op Code
2	OPA	Variable A Operand
3	OPB	Variable B Operand
4	5A	INC Op Code
5	OPC	Variable C Operand
6	52	MUL Op Code
7	OPD	Variable D Operand
8	OPE	Variable E Operand
9	51	SUB Op Code
10	53	DIV Op Code
11	OPF	Variable F Operand
12	59	NEG Op Code
13	OPG	Variable G Operand
14	OPH	Variable H Operand
15	55	Logic OR Op Code
16	50	ADD Op Code
17	54	Logic AND Op Code
18	5D	MOV Op Code
19	OPY	Results Data Field (Y)