Technical Memorandum 4232

A Personal Computer-Based, Multitasking Data Acquisition System

(NASA-TM-4232) A PERSONAL COMPUTER-BASED, MULTITASKING DATA ACQUISITION SYSTEM (NASA) 15 p CSCL 09B

N90-29107

Unclas

H1/61 0305024

NASA Technical Memorandum 4232

A Personal Computer-Based, Multitasking Data Acquisition System

Steven A. Bailey
Goddard Space Flight Center
Wallops Flight Facility
Wallops Island, Virginia

NASA
National Aeronautics and
Space Administration
Office of Management
Scientific and Technical
Information Division

1990

Summary

A multitasking, data acquisition system has been written to simultaneously collect meteorological radar and telemetry data from two sources. This system is based on the personal computer (PC[1]) architecture. Data is collected via two assynchronous serial ports and is deposited to disk. The system is written in both the "C" programming language and assembler. It consists of three parts: a multitasking kernel for data collection, a shell with pull down windows as user interface, and a graphics processor for editing data and creating coded messages. An explanation of both system principles and program structure is presented.

Introduction

In use today are many systems used to track upper atmospheric conditions. This knowledge can be used for many things, including guidance information necessary for sounding rockets. A system that integrates balloon radiosonde and radar tracking information into one data set currently exist. This system was written by Optimetrics, Inc. of Ann Arbor, Michigan. It incorporates one of several balloon radiosonde systems with a modified Enterprise Electronics WF-100 radar. At present, the system works but is not portable between sites. This is due to noncentralized modifications to software, along with hardware changes and additions. System documentation is scarce, and program modularity is nonexistent.

The author's task was to make a more usable system. This would involve rewriting a major portion of the original code or rewriting the entire system. After some study, it was decided to start from scratch. A complete new data system would be written. The only restrictions were that it had to run on any PC and data had to be gathered via assynchronous serial ports. The net result must show graphical progress of data collection with final creation of one of several meteorological coded messages.

This paper describes all aspects of this new and improved data system. System ideals and principles are examined. Reasons for chosen design are explained. Program flowcharts and detailed descriptions of program structure are given. Finally, the creation of coded messages is explained in full.

System Ideals

This data system performs the same basic functions as any data system. Data is collected and deposited to a storage device while giving the user status information. The most important aspect of this process is ensuring data integrity. Many things must be considered to maintain this directive. The most important of these are data rate and data medium. A high data rate usually requires a special data medium to ensure proper throughput. A low data rate can be maintained with a simple data medium.

The data medium used for this system is assynchronous RS-232. This is a simple medium to implement on most micro-computers. Fortunately, any PC can be easily populated with two such devices.

Principles

This data system is based on several tightly coupled interrupt service routines. These are programs which run independent of one another and independent of any foreground program. Interrupt service routines are demand-driven. When data is available, these routines are called directly from the hardware level. When data is not available, these routines sit idle. No program intervention (like polling) is ever needed when using interrupt routines. They support themselves.

Through the use of interrupt service routines, multitasking is available. While data is collected and deposited by interrupt routines, a program (any program) can run simultaneously. This program is the user shell. It runs in the foreground while the interrupt routines run in the background. Since the interrupt routines are demand-driven, they get as much CPU time as needed to service their particular needs. Any time left over is given to the foreground program. We call this method of operation demand-driven or nonpreemptive multitasking. Figure 1 is a block diagram of the data system.

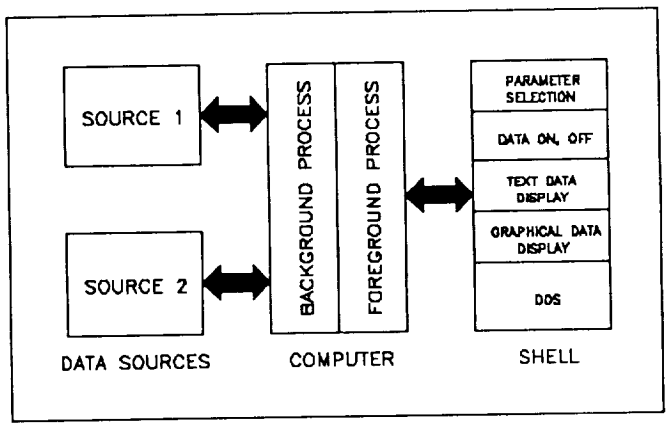

Figure 1. Data system block diagram.

Hardware

On the PC platform, there are two types of interrupts. The

first type is a software interrupt. This type is an instruction that makes the cpu jump directly to a given routine. The second type is hardware interrupts. Of these, there are two varieties. The internal variety is usually confined to the system or motherboard. These hardware interrupts are dedicated to interrupting the cpu when catastrophic events take place. A memory error is a good example of an internal hardware interrupt. The external variety of hardware interrupts is found from external devices. Hardware interrupts are responsible for most of the communication which occurs between peripheral boards and the motherboard.

A lower part of computer memory is dedicated to what is called an interrupt vector table. This is where a list of addresses point to potential interrupt service routines. Much of this table is filled with addresses DOS needs to perform routine operations. Some examples of these are keyboard entry, disk I/O, video services, etc. See Figure 2 for an example of an interrupt vector table.

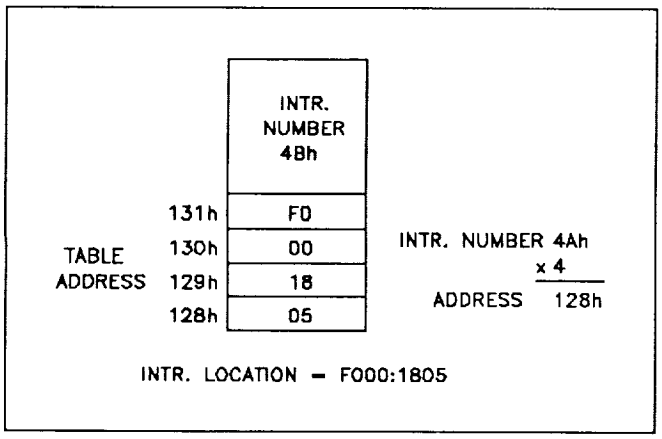

Figure 2. Interrupt vector table.

There is a chip native to all PCs called the 8259A programmable interrupt controller. This chip coordinates all hardware interrupts channeled to the cpu. With this chip, priority can be assigned to all interrupts. This is necessary, because some interrupts must have high priority. An example would be the system clock. It has the highest priority, because event timing is critical. With this chip, incoming interrupts can be made available or unavailable to the cpu. Think of the 8259A as an intelligent switch between hardware interrupts and the cpu.

Another chip native to all PCs is the 8254 programmable interval timer. This chip has 3 clocks that are driven by a crystal oscillator at 1.192 Mhz. Each clock is independently programmable with a 16-bit word. This word is used as a countdown timer. Whenever this word counts down to 0, an event occurs. Of the 3 clocks, we only use clock 0. This clock is used to drive a hardware interrupt line called IRQ0. This is the interrupt of highest priority we mentioned previously.

Whenever clock 0 counts down to 0, an interrupt is posted on the IRQ0 line. Since the control word responsible for clock 0 is set at 65535, an interrupt occurs at 18.2 hertz. This number is found by dividing 1192180 by 65535. The system uses this interrupt to maintain 32-bit time. We use this interrupt as a trigger for our data system. Figure 3 shows the relationship between hardware interrupts and interrupt service routines.

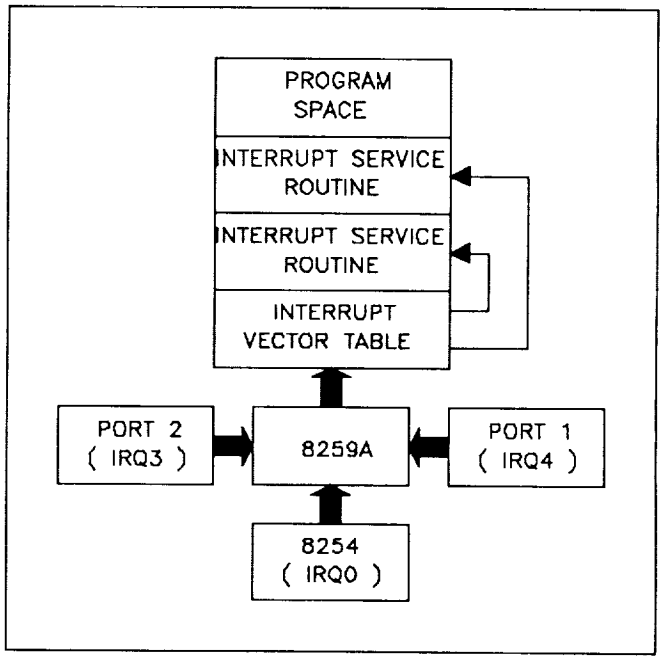

Figure 3. Interrupt flow.

Design

Many considerations were made when designing this new data system. This system had to run on any PC. This includes all PCs with the 8088, 80186, 80286, and 80386 microprocessors. System memory of not more than 640 kbytes would be available. Easy user installation was a must. Data throughput would not be high.

Microsoft OS/2[2] was considered for its multitasking environment. Unfortunately, it will not run on anything less than an 80286. It also requires a minimum of 2 megabytes of memory, a tedious installation procedure, and is not guaranteed to run on any PC clone.

Microsoft Windows[2] was also considered for its graphic user interface (GUI). It too must run on a 80286 or better. It is slow, memory hungry, and is less than intuitive to use. Installation, although not as bad as OS/2, is quite lengthy.

A multitasking kernel called AMX[3] was considered. It has no user interface, but does allow for multitasking on all PCs. It was found to offer an archaic programming interface, which made it very difficult to program. Bad remarks were also given AMX from programmers more familiar with the system.

Another multitasking kernel called OMNIVIEW[4] was considered. This system does include a programming interface (API), but there is no documentation. Since the API consists of approximately 150k of assembler source code, the author thought it wise to seek other avenues.

Since no operating system or system enhancement seemed to fit what I was looking for, I decided to write my own multitasking kernel. It only needed to multitask at most two comm ports with one foreground program. Although not a trivial task, it proved a worthwhile effort in better understanding the inner workings of DOS.

I also decided to write a pull-down window user interface. A system was needed where the user could access many variables and processes with a minimum of confusion. Many commercial DOS programs use this method and are quite effective.

Programs

As mentioned, this new data system is basically divided into three parts. The heart of the system is the multitasking kernel. This is where data is collected, reformatted, and deposited to disk. It is written entirely in Borland Turbo Assembler[5] using their Turbo C[5] interface structure. This not only makes this interface clean, it also makes the assembly code easier to read.

Sitting on top of the kernel is the user interface or shell. It is here that the user interacts with the system. Variables needed to drive the kernel are altered here. Information relating to port assignments and data structure, protocol, and handshaking can be entered, as well as station weather information, time, and location. Finally, the graphics processor is "launched" or run from the shell. All information driving this graphics processor can be edited from the shell.

The final program is the graphics processor. It is of the stand-alone variety. It can be run from the user shell or from the DOS prompt. The graphics processor reads a file as its data source. This file can be read before, during, or after data collection. Due to the multitasking nature of this system, the graphics processor can read a data file while it is being written. To the graphics processor, this is just a file. This offers the advantage of using the graphics processor to process data at any time.

Multitasking Kernel

Kernel interrupt service routines

The heart of this data system is contained in the kernel. It is here that data is captured, verified, and deposited to disk. The kernel runs in the background independent of any foreground program. The only direct link to the kernel is via the user shell discussed in the next section.

For all practical purposes, the kernel consists of two types of routines. The first type is the interrupt service routines. Of the six routines, two are dedicated to the comm ports. The other four routines enable the kernel to write to disk using DOS services. The second type of kernel routines is general purpose in nature. These kernel routines either support the comm port interrupt service routines or it allows for initialization and de-initialization of the system. Since assembler routines can be less than intuitive to follow, each routine from the kernel will be explained in detail. Figure 4 is a block diagram of the kernel.

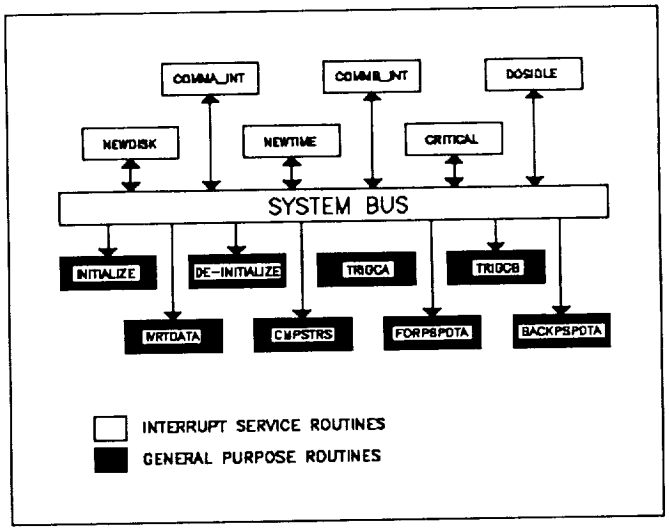

Figure 4. Kernel block diagram.

Interrupt service routines **comma_int** and **commb_int** are essentially the same. Each uses its own set of local variables, but their functions are identical. Each routine

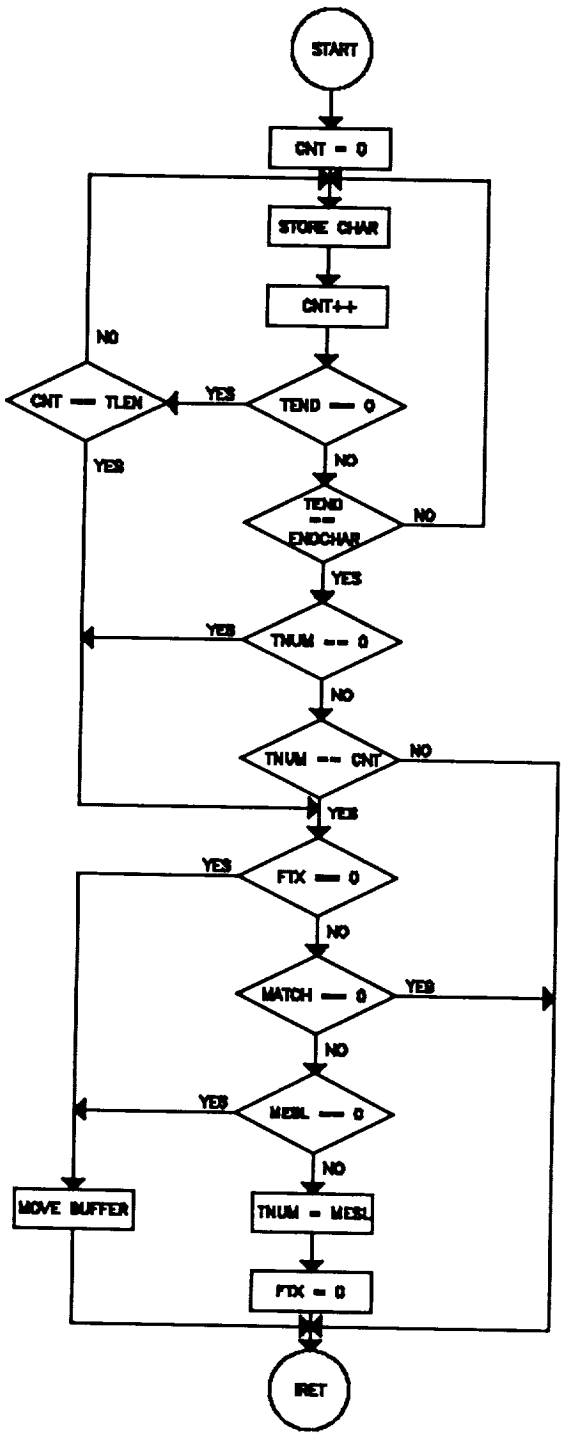

Figure 5. Comm port interrupt flowchart.

services data coming in from its respective port. Upon an interrupt, these routines input the current byte latched on the port. This byte is then saved in a local buffer. What follows is a series of tests, which prove or disprove data validity. The extent of these tests is dependent on the state of variables programmable from the user shell. There are a total of eight different tests that can be programmed into each comm port service routine. Figure 5 is a flowchart of the comm port interrupt service routines.

There are five variables that control data validity for each comm port. Figure 6 is a pseudo truth table of those variables and their relation to one another. For this map, states are on when a given variable has a value greater than zero. Since this not a true truth table, only eight conditions or tests are possible.

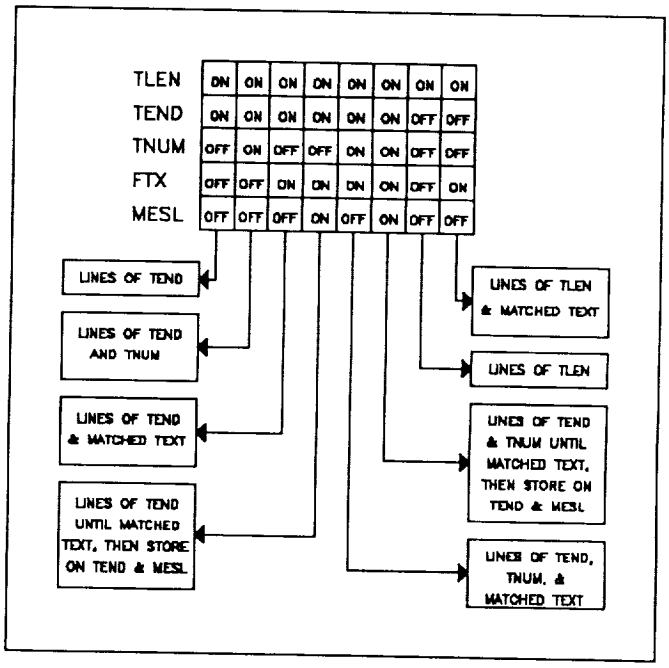

Figure 6. Pseudo truth table.

The **dosidle** interrupt service routine is activated whenever DOS enters an "idle" or "quiet" state. A good example of this state is when a program waits for keyboard input. When this state occurs, the software interrupt 28-hex is repeatedly called by DOS. This particular interrupt service routine is actually inserted before the original dosidle routine. When the system goes "idle", a 28-hex interrupt is initiated, and this particular routine is called. From this routine, the original dosidle routine is called, and a return is made to this routine. From here, four checks are made. These checks determine whether data should be written to disk.

From Figure 7, time is the first check made before data deposit. At least 6 seconds must have passed since the last

write. This value was chosen as a compromise between data buffer space and foreground update rate. Next, the last write must be complete for a current write to occur. Data must be available in the buffer and finally, the disk must not be busy. When all the above criteria are met and DOS is idle, a write will occur.

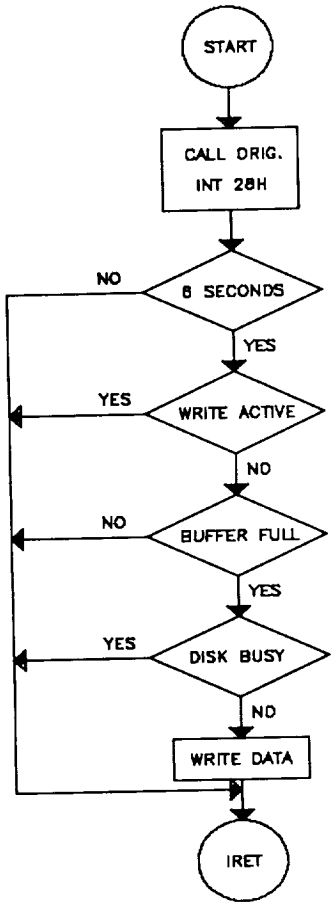

Figure 7. Dosidle interrupt flowchart.

The **newdisk** interrupt service routine is called whenever DOS reads or writes to a device. This device must use the software interrupt 13-hex services. This includes all DOS standard floppy and hard drives. As with the dosidle routine, this routine is inserted before the original interrupt 13-hex routine. The purpose of this routine is simple. It sets a disk active flag before the original routine is called. When a return is made from the original routine, the flag is cleared. This flag is used throughout the kernel. It indicates when the disk is busy. Figure 8 is a flowchart of the newdisk interrupt service routine.

The **newtime** interrupt service routine is called whenever the 8254 timer chip initiates an interrupt. The default rate is

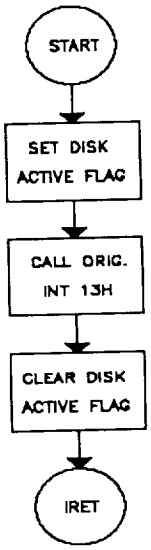

Figure 8. Newdisk interrupt flowchart.

18.2 hertz. As shown, newtime is inserted before the original interrupt 08-hex routine. The original routine updates a 32-bit memory location with current time. Newtime is used to write data to disk and initiate interrupts to comm ports if that function is enabled. Like the dosidle routine, criteria must be met before a write can be made. In addition to the four criteria described in dosidle, another condition must be met. DOS cannot be busy when a background write is needed. When in dosidle, DOS is being used to call dosidle, so this condition need not be checked.

Within newtime, a disk write is made when the 5 criteria are met. Whether met or not, a variable called wrtime is decremented if not equal to 0. This variable is used as a local timer for the kernel. When this variable reaches 0, enough time has elapsed for a write to be considered. This variable remains zero until a write occurs. This way, every time newtime is subsequently entered, a write will be considered.

The last thing to occur before newtime exit is to check whether signals or interrupts should be sent to the comm ports. If they should be sent, the current rate of interrupt is compared with the current elapsed time from last interrupt. The current rate of interrupt is a variable altered from the user shell. See Figure 9 for a flowchart of newtime.

Finally, the last interrupt service routine of this system is called **critical**. It replaces the original interrupt 24-hex routine with a single interrupt return "iret" instruction. The original routine services critical errors. An example of a critical error is trying to print to a device that does not exist.

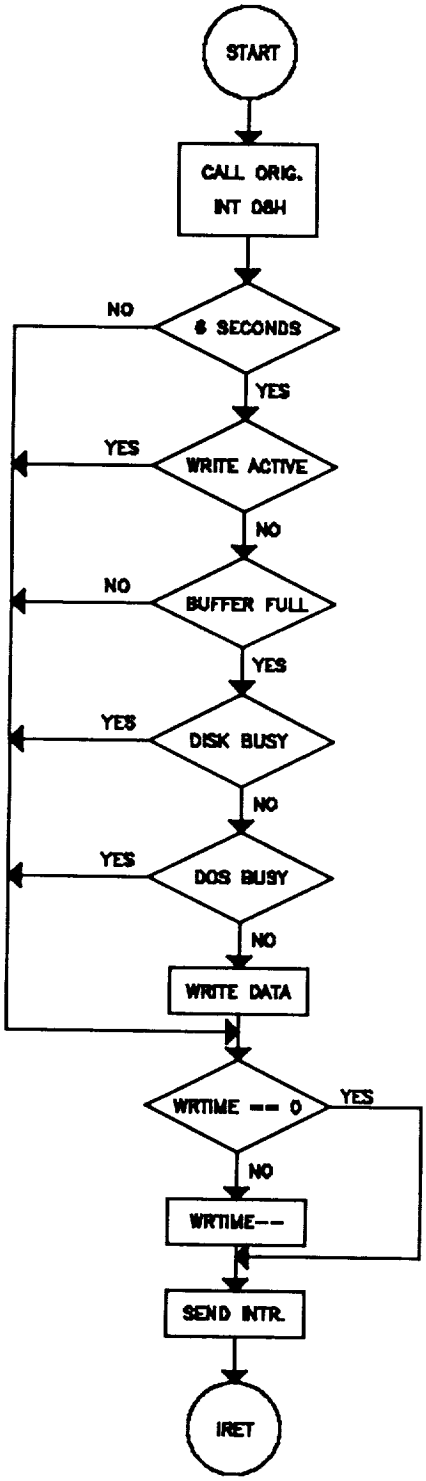

Figure 9. Newtime interrupt flowchart.

The original critical error routine would halt all programs and display an obscure message. Critical simply keeps programs from halting when critical errors occur.

Kernel general purpose routines

Wrtdata is a general purpose routine called from interrupt service routines newtime and dosidle. Its purpose is to write data to the chosen disk drive. Before data is written, this routine must switch to the background PSP and DTA. These contain DOS environment information necessary for a background write. After data is written, the file is committed. This means the directory information defining that file is updated. In the event of a power failure, all data will exist in that file. Only data not committed will be lost. Since data is written every 6 seconds, only a maximum of 6 seconds of data can ever be lost. An active flag variable is set and reset upon entry and exit from wrtdata. This variable is used throughout the kernel as a flag. When this flag is set, DOS is busy writing data. Figure 10 is a flowchart of wrtdata.

Trigca and **trigcb** are essentially the same general purpose routine. Each is dedicated to a user defined comm port. Upon command from interrupt service routine newtime, these routines output a designated byte to the chosen port. This can serve as a signal or trigger to an external device. In the case of this data system, these routines serve as a periodic trigger defined from the user shell.

Cmpstrs is a general purpose routine called from both comma_int and commb_int interrupt service routines. Cmpstrs tries to find a match between a string of characters and a buffer. When a match is found, a value of 1 is returned in the "dl" register. This function is necessary in completing several of the tests when checking data validity.

Routines **forpspdta** and **bacpspdta** both perform the inverse of each other. Forpspdta switches from the background PSP and DTA to the foreground PSP and DTA. This is necessary when control is given back to all foreground programs. Likewise, bacpspdta switches from the foreground PSP and DTA to the background PSP and DTA. This is only necessary when the background routine, wrtdata, writes data to disk.

Kernel initialization routines

The following routines are called via the user shell. They are called in the sequence that follows. Their combined purpose is to start or initialize the data system. See Figure 11 for a block diagram of initialization routines.

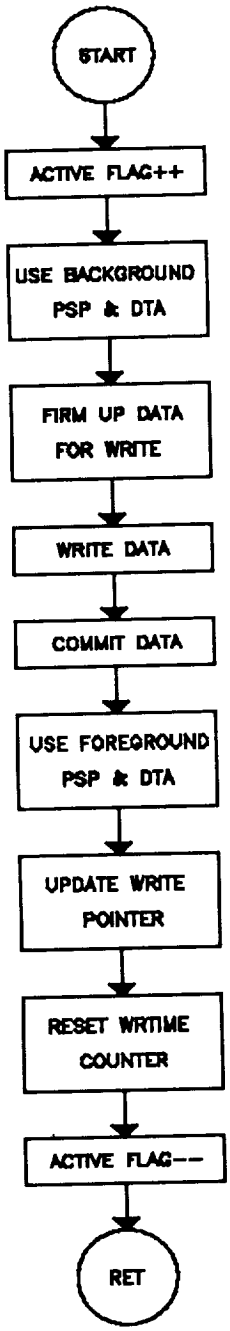

Figure 10. Wrtdata flowchart.

_curpspdta saves the current program segment prefix (PSP) and disk transfer area (DTA). This information is needed whenever a background disk write is made. The shell and kernel share the same PSP and DTA. Whenever another instance of "command.com" is run, a new PSP and DTA is issued. To carry out background disk writes, the PSP and DTA that curpspdta saves must be used.

_initial has two purposes. First, it initializes all data pointers to their appropriate buffers. Second, it creates the proper header information pertaining to data type, time type, time length, and local buffer count. In essence, this routine initializes all parameters used for buffer manipulation.

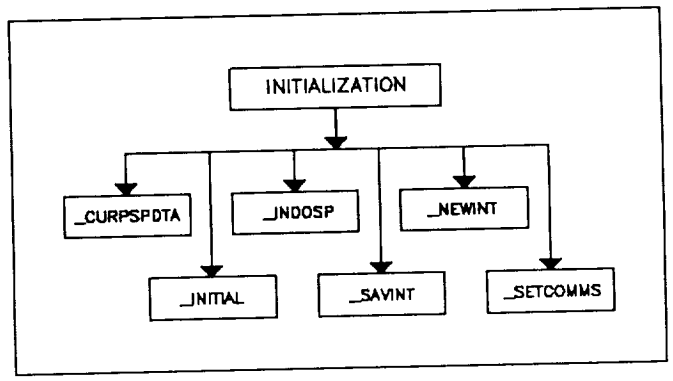

Figure 11. Initialization block diagram.

_indosp simply captures the address of the variable that indicates whether DOS is busy or not. This address is used throughout the kernel.

_savint saves original interrupt vectors. This includes the system clock (int 08-hex), the disks (int 013-hex), the critical error flag (int 024-hex), dosidle (int 028-hex), and both comm ports.

_newint changes all the above interrupt vectors to new vectors. These vectors now point to our interrupt service routines.

_setcomms sets up both comm ports for interrupt-driven service.

Kernel de-initialization routines

The following routines are also called via the user shell. They are called in the sequence which follows. Their combined purpose is to stop or de-initialize the data system. See Figure 12 for a block diagram of de-initialization routines.

_rsetcomms disables both comm ports from interrupt-driven service.

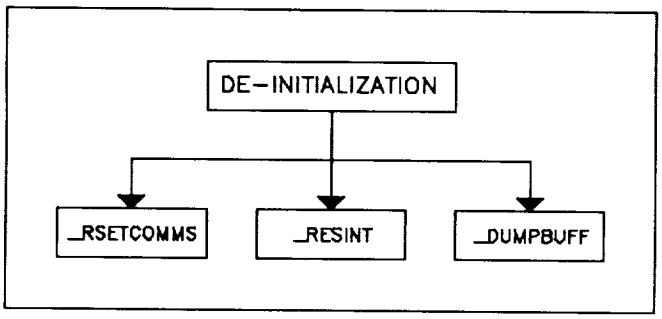

Figure 12. De-initialization routines.

_resint changes all data system interrupt vectors back to their original state.

_dumpbuff empties the data queue of remaining data. This data is written to disk.

User Shell

Program structure

The user shell is written in the "C" programming language and is designed in a top down fashion. Figure 13 is a tree diagram of the shell structure. There are three levels to this structure enclosed in the dotted line. The first level is responsible for initializing, running, and de-initializing the shell. The second level dispatches keyboard commands. The third level runs alternate programs, starts and stops the data system, manipulates the windows, and manipulates all variables. The following is a brief description of each level.

Level 1

During **initialization**, many things occur. Filenames given on the command line are opened and video type is determined. Windows and their supporting data structures are read in. Keyboard setup buffers are read in and stored. Finally, window data structures are further enhanced for proper operation.

The shell begins during the **run** phase of level 1. This will be explained in level 3.

De-initialization prepares for proper system shutdown. If the data system is active, it is switched off, then any remaining data is written. All parameters previously read in are saved to disk.

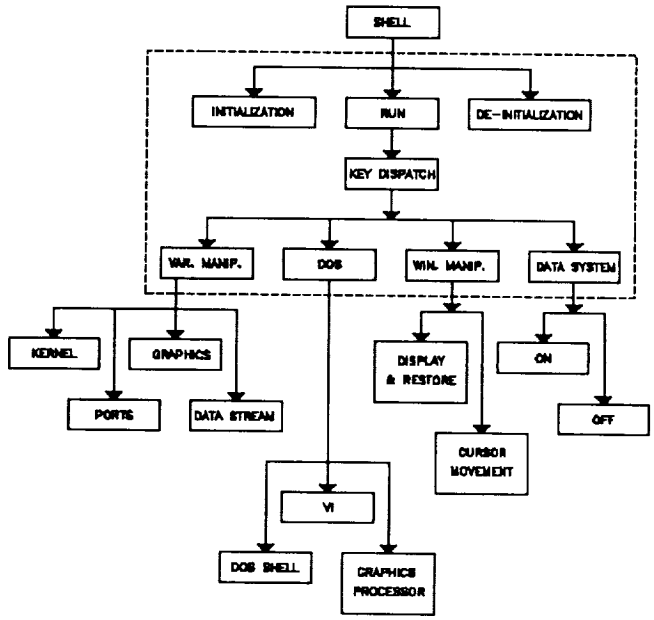

Figure 13. Shell structure.

Level 2

This level is managed by the **keyboard dispatcher**. This dispatcher accepts keyboard input and performs the appropriate action. These actions are described in level 3.

Level 3

Window manipulation is responsible for two things. **Cursor movement** is first achieved here, followed by **window display and restore**.

Under the category of **variable manipulation**, all variables associated with the shell are available for update. This includes variables that drive the **kernel**, variables that support the comm **ports**, variables that supplement the **data stream**, and variables that initialize the **graphics processor**.

The **data system** is turned **on** and **off** through the shell. When turned on, a file is opened for networked read and write. The kernel is then initialized for interrupt-driven service. When the system is turned off, the reverse occurs.

Finally, the **DOS** category is responsible for running another version of itself. This is done to run DOS programs from the command line or run DOS programs directly from within the shell. Two examples of the later are the **graphics processor** and the **vi** editor. The **vi** editor is used to edit parameters found in a given **graphics processor** parameter file. The

graphics processor is used to process incoming data graphically.

Graphics Processor

Program philosophy

Like the user shell, the graphics processor is written in the "C" programming language and is designed in a top down fashion. Since it is a stand-alone program, it can be run independently of the user shell. For this discussion, we will reference the graphics processor when launched from the shell.

Like any data processing program, the graphics processor is responsible for reading and displaying data. Also, like any data processing package, the graphics processor enables the user to apply editing and smoothing functions to data. However, unlike most data processing programs, this processor does all the above via a graphic display. Figure 14 is a tree diagram of the program structure.

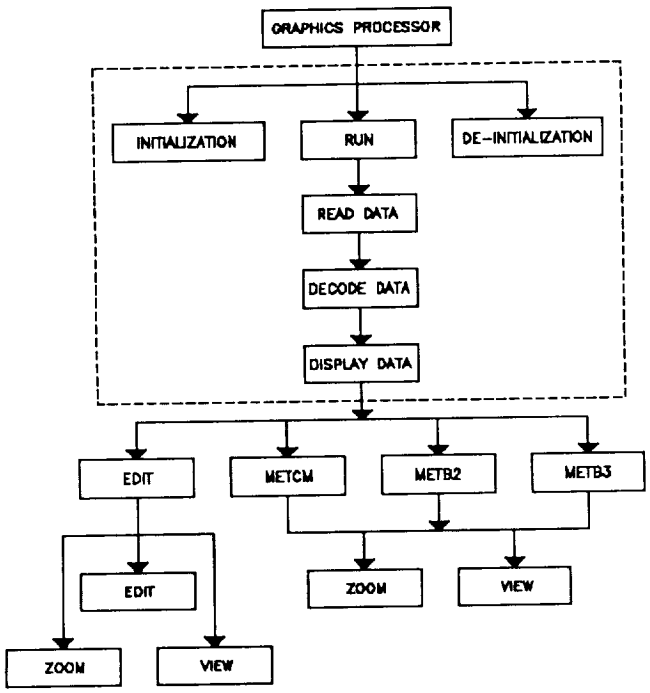

Figure 14. Graphics processor structure.

Program structure

The graphics processor is structured into four levels. The first level is very similar to that of the shell. It is here that initialization, running, and de-initialization of the graphics processor occur. The second level is solely responsible for reading the appropriate data file. The third level takes that data and decodes it. Finally, decoded data is displayed graphically in level 4. During this display, the user is allowed to manipulate the data via graphic controls. The following is a brief description of each level.

Level 1

The graphics processor is a general purpose program. It can be customized by the user to display and process data of choice. This is accomplished through a graphics parameter file. This file is ascii and contains many graphics primitives that describe to the processor what needs to be done. During **initialization**, this file is read in, and many variables supporting those graphics primitives are initialized.

Once initialization is complete, the graphics processor is started. It is here that data is read in and displayed. After the user exits from the processor, a **de-initialization** procedure occurs. All files formally opened are closed, and a return is made to the shell or DOS.

Level 2

This level has one function. The chosen data file is read in. This can be a data file currently being written by the kernel, or it can be any disk file of the correct format. This format is called the Type Length Data (TLD) format. Data is recorded in logical blocks defined by the user. As seen in Figure 15, this format has three distinct fields.

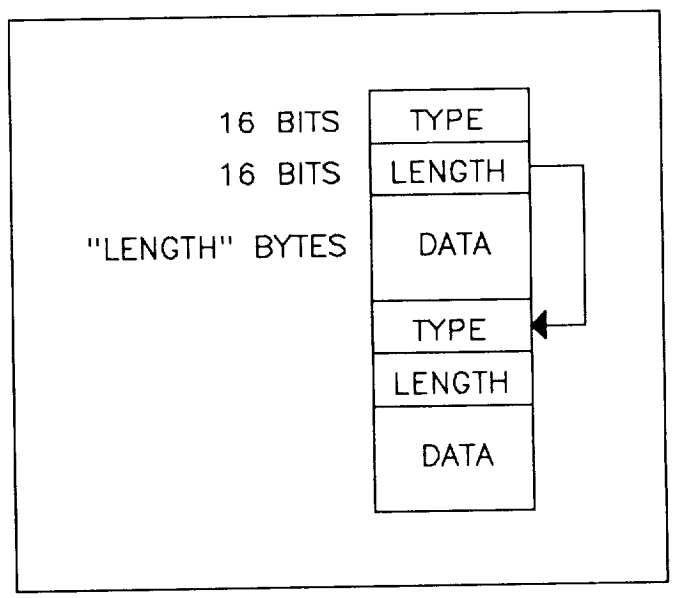

Figure 15. TLD format.

The first field is a 16-bit word called "type." This is a number chosen by the user, to represent data. The second field is a 16-bit word called "length." This is the number of bytes found in this block of data. The third field is the actual data. The amount of data here is "length" in bytes.

This data structure is a modified linked list. It enables a program to quickly read through a data file in search of a particular data "type." Since "type" is a 16-bit word, there are 65,536 data types available at any given time. Likewise, "length" is also a 16-bit word. This means a data block can be up to 65,536 bytes in length. The TLD format lends itself to flexibility. When data becomes available, it can inserted into the data stream without regard to other data. The user only needs to keep track of data "type" numbers.

Level 3

Once data is read in, it must be **decoded**. It is here that data "type" is matched to a particular decoding scheme. Data is decoded and placed into a structure. This structure is used throughout the program. It contains many variables, including temperature, pressure, relative humidity, time, wind speed, etc. These variables are updated every time a data "type" is read in. Those variables not found in a particular data "type" are initialized to an "empty" state.

Level 4

The above data structure is passed to a part of the program that **displays data**. Although there are many varieties of display that can be defined using the graphics parameter file, every display has one of four filters it must pass through.

The **EDIT** filter allows the user full command of the processor. With this filter turned on, one can **view** data via a graphics screen. **Zooming** in on data is possible using the keyboard cursor. Finally, bad data can be **edited** out also using the cursor controls.

The three filters **METCM**, **METB2**, and **METB3** are used to create coded messages. The same display is shown when **viewing**, with the superposition of zone level lines when the y-axis is expressed as height. **Zooming** is still allowed but **editing** is not. The final message is not written until the graphics processor is exited.

Coded Messages

Meteorological (MET) messages are a condensed form of current atmospheric conditions. For this system, a rising balloon is tracked, telemetry data is collected, and a series of calculations are made. For MET messages, the atmosphere

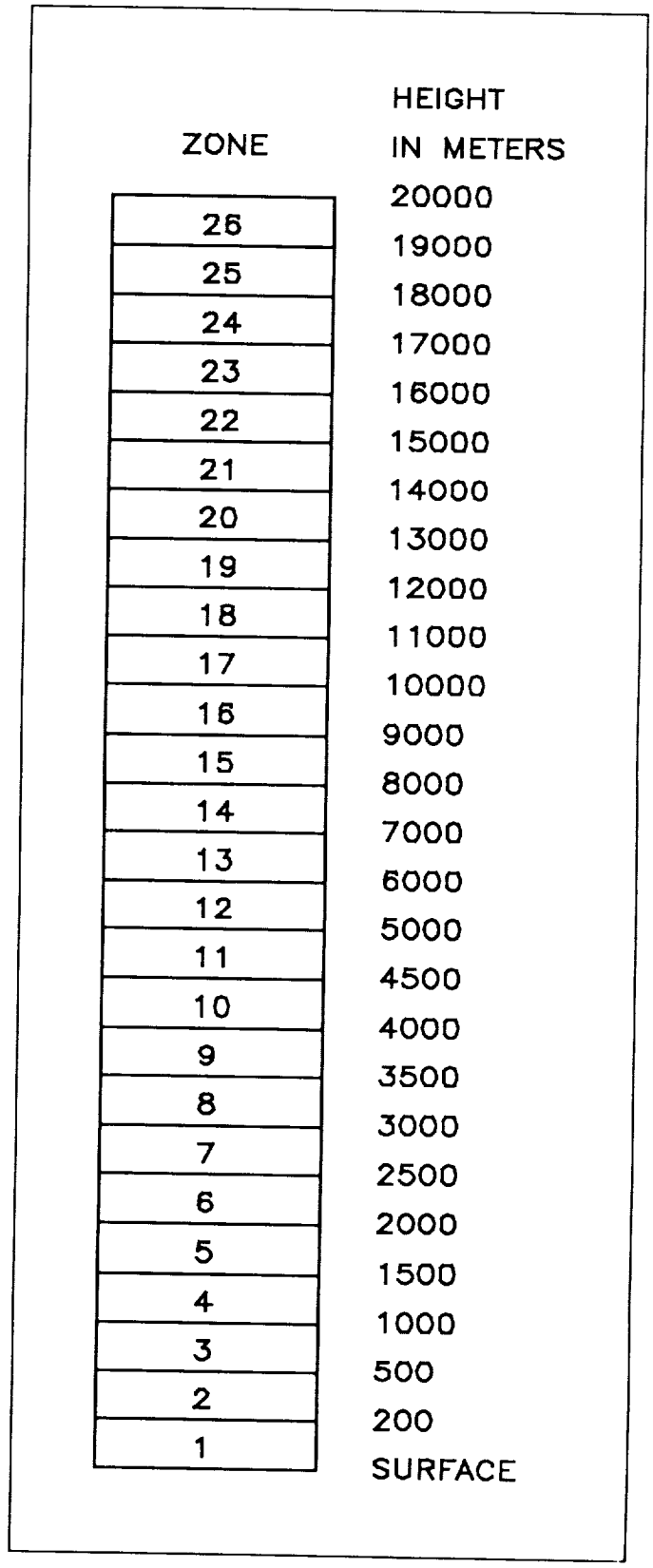

Figure 16. METCM zone distribution.

is divided into layers or zones. Each zone is a particular thickness, depending on the MET message used. See Figure 16 for an example of zone distribution.

METCM

For the Standard Artillery Computer Meteorological Message (**METCM**), there are 26 zones of interest. In each zone, 4 calculations are made. They are mean wind speed, mean wind direction, mean virtual temperature, and mean pressure. As can be expected, the mean values for each calculation are found by taking the average of all readings within a zone.

Wind speed and direction are first combined to form vectors in the **x** and **y** plane. For the expressions

$$x = r \cos \theta, \qquad y = r \sin \theta,$$

where

r = wind speed
θ = wind direction,

the values of **x** and **y** are calculated and summed for each zone. Their average is then used to find the mean wind speed or magnitude and the mean wind direction or angle.

$$magnitude = \sqrt{x^2 + y^2}$$
$$ang = \arctan\left[\frac{x}{y}\right]$$

Before a mean virtual temperature can be found, the current values of temperature must be converted to virtual temperature. This is simply the conversion of temperature in centigrade to temperature in kelvin. The constant **273.16** is first added to each value of temperature before the sum and subsequent average is made across a zone.

METB2

For the Standard Ballistic Meteorological Message (Surface to Air - **METB2**), there are 15 zones of interest. Again, there are 4 calculations made for each zone. They are ballistic wind direction, ballistic wind speed, ballistic temperature, and ballistic density. Instead of finding mean values, as with **METCM**, these calculations are dependant on atmospheric weighting factors and previous calculations.

As in the **METCM**, the mean wind speed and direction are found for each zone using vector notation. Virtual temperature is found using the following equations:

$$virtual\ temp = \frac{ktmp}{1 - E\left[\frac{lapse}{pres}\right]},$$

where

ktmp = **current temperature in celsius + 273.16**
E = **0.379**
pres = **current pressure in millibars**
lapse = **lapse rate**

$$lapse\ rate = rh \cdot A \cdot e^{\left[\frac{B \cdot temp \cdot C}{D + temp}\right]}$$

rh = **relative humidity**
A = **0.0611**
B = **7.5**
C = **2.3025**
D = **237.3**
temp = **current temperature in celsius.**

Density is found using the following equation:

$$density = \frac{F \cdot pres}{virtual\ temp},$$

where

F = **348.38.**

A mean virtual temperature and mean density are then calculated for each zone. Following this, each value is converted to a percent-standard. For each zone, there is a known standard temperature (in kelvin) and density (in gm/m^3). These standard values are divided into the mean values to determine a percentage of the following form:

$$percent\ standard = \frac{mean\ value \cdot 100}{standard\ value}.$$

At this point, four tables are created containing mean wind **x** vector, mean wind **y** vector, standard temperature, and standard density for every zone. What follows is matrix multiplication using the four tables versus four matrices of weighting factors.

These weighting factors are used to determine cumulative values for each zone. For example, the **METB2** has 15 zones. For each zone, a number (using virtual temperature) represents the cumulative effects of temperature up to and including that particular zone. Since there are 15 zones, there are 15 numbers, each representative up to their

particular zone.

Before the cumulative effects at each zone are calculated, the following matrix multiplication must occur:

Where $\mathbf{K_i}$ represents a diagonal matrix of zone values, and $\mathbf{M_{ij}}$ represents an upper diagonal matrix of weighting factors.

The matrix product will be in Q_{ij}.

$$\begin{bmatrix} k_1 & 0 & 0 & 0 \\ 0 & k_2 & 0 & 0 \\ 0 & 0 & k_3 & 0 \\ . & . & . & . \\ 0 & 0 & 0 & k_n \end{bmatrix} \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{1n} \\ 0 & m_{22} & m_{23} & m_{2n} \\ 0 & 0 & m_{33} & m_{3n} \\ . & . & . & . \\ 0 & 0 & 0 & m_{nn} \end{bmatrix}$$

To find the cumulative effects at each zone, the following summation must occur:

$$Z_i = \sum_{j=1}^{n} Q_{ij} \, .$$

It is these values of $\mathbf{Z_i}$ that form the final coded message.

METB3

For the Standard Ballistic Meteorological Message (Surface to Surface - **METB3**), all processes are identical to those used with **METB2**. The only difference is each message uses a unique set of weighting factors for winds, temperature, and density.

Concluding Remarks

This new data system has been tested in the field and does work as designed. Due to its modular construction, all or parts of this system have been used to develop three subsequent data systems based on the PC architecture. As was found, the PC serves as a powerful and inexpensive computer on which to base a low rate data system.

References

1. PC is a trademark of International Business Machines Corp.

2. OS/2 and Windows are trademarks of the Microsoft Corp.

3. AMX is a trademark of Kaeak LTD.

4. OMNIVIEW is a trademark of Sunny Hill Software.

5. Turbo Assembler and Turbo C are trademarks of Borland International.

Report Documentation Page

1. Report No. NASA TM-4232	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle A Personal Computer-Based, Multitasking Data Acquisition System		5. Report Date September 1990
		6. Performing Organization Code 972
7. Author(s) Steven A. Bailey		8. Performing Organization Report No. 90B00133
		10. Work Unit No.
9. Performing Organization Name and Address NASA Wallops Flight Facility Wallops Island, VA 23337		11. Contract or Grant No.
		13. Type of Report and Period Covered Technical Memorandum
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washingtion, DC 20546-0001		14. Sponsoring Agency Code

15. Supplementary Notes

Steven A. Bailey: Wallops Flight Facility, Wallops Island, Virginia.

16. Abstract

A multitasking, data acquisition system has been written to simultaneously collect meteorological radar and telemetry data from two sources. This system is based on the personal computer (PC) architecture. Data is collected via two assynchronous serial ports and is deposited to disk. The system is written in both the "C" programming language and assembler. It consists of three parts: a multitasking kernel for data collection, a shell with pull down windows as user interface, and a graphics processor for editing data and creating coded messages. An explanation of both system principles and program structure is presented.

17. Key Words (Suggested by Author(s)) Multitasking Data acquisition system PC	18. Distribution Statement Unclassified--Unlimited Subject Category 61

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of pages 16	22. Price A03

NASA FORM 1626 OCT 86

NASA-Langley, 1990