LBL-24707 c.2

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

Presented at the Fifth Annual Meeting of the Society of Magnetic Resonance in Medicine, Montreal, Quebec, Canada, August 19–22, 1986, and to be published in IEEE Transactions on Medical Imaging

An Instrument Control and Data Analysis Program for NMR Imaging and Spectroscopy

M.S. Roos, R.A. Mushlin, E. Veklerov, J.D. Port, C. Ladd, and C.G. Harrison

January 1988

Donner L

Biology & Medicine Division

Prepared for the U.S. Department of Energy under Contract DE-AC03-76SF00098

An Instrument Control and Data Analysis Program for NMR Imaging and Spectroscopy

Mark S. Roos[*], Richard A. Mushlin[†], Eugene Veklerov[*], John D. Port[†], Carol Ladd[†], and Colin G. Harrison[†]

[*]Lawrence Berkeley Laboratory,
University of California, Berkeley, CA 94720

[†]IBM Corporation, Armonk, NY 10504

This work was supported by the IBM Corporation.

Address correspondence to:
Mark S. Roos
Lawrence Berkeley Laboratory
1 Cyclotron Rd. MS 55-121
Berkeley, CA 94720

Abstract

We describe a software environment created to support real-time instrument control and signal acquisition as well as array-processor based signal and image processing in up to five dimensions. The environment is configured for NMR imaging and *in vivo* spectroscopy. It is designed to provide flexible tools for implementing novel NMR experiments in the research laboratory. Data acquisition and processing operations are programmed in macros which are loaded in assembled form to minimize instruction overhead. Data arrays are dynamically allocated for efficient use of memory and can be mapped directly into disk files. The command set includes primitives for real-time control of data acquisiton, scalar arithmetic, string manipulation, branching, a file system and vector operations carried out by an array processor.

Keywords: NMR instrumentation, real time software, signal processing, image processing

1 Motivation

Modern instrument systems rely on computers for nearly every aspect of their operation. Complex instruments are characterized by large sets of control parameters, intricate operation sequences, critical timing requirements, diverse data structures, and highly cooperative functional elements with multiple processors. The control software stands between the user and this inherent complexity, and must minimize demands on the user while maintaining access to the full capabilities of the instrument system. In a research environment it is essential that limits be set by the inherent characteristics of the hardware and not by the forsight of the application programmer. This paper describes a software environment in which the degree of complexity, as seen by the user, can be tailored to that user's individual needs and level of expertise.

Most application software is run by commands. The functionality of a program is often described in terms of the commands available. For control of complex instrument systems, however, it is impractical to have a separate command for every combination of operations that the system can perform . Thus the command set usually consists of simple operations and a method of grouping sequences of commands together, saving the group under a single name, and executing the group as a whole. These command groups are called by various names including procedures, automation sequences, and macros. The term macro will be used henceforth.

The performance and personality of a program of this type are determined to a large degree by the command set and the sophistication of the data flow and execution control in macros. Traditional approaches to the design of instrument control programs have often had several of the characteristics listed below:

- Large command sets contain individual commands to implement sophisticated data acquisition and processing tasks.
- Interpretive executives control the execution of commands in a macro.
- Parameters are passed by value to a macro when it is run and substituted for dummy symbols.
- Data are stored in static structures under control of the program.
- Variables internal to the program are not acessible to the user for computation and program control.

The program described herein has somewhat different characteristics:

- The command set is smaller, consisting of commands that perform primitive functions, so that several may need to be grouped together to perform the task of one command in a program with a large command set.
- The macros are processed when they are loaded into the computer so that all symbols (commands and arguments) are resolved into addresses. This pre-processing, called

assembly, greatly increases the speed of the executive at run-time. Unlimited nesting of macros is allowed.

- Parameters are passed to macros by reference (that is, the address is passed). This is consistent with the assembly strategy, and allows large data structures to be passed efficiently among macros.
- Data structures (arrays) are dynamically allocated under user control.
- Internal variables used to describe the state of the hardware are available to the user, although their values may only be changed by executing approproate commands. These 'hooks' into the control program are useful for controlling branching in macros, writing macros that interact with the graphics display, and for documentation such as header files and display annotation.

The functionality of a complex command set can be equalled or exceeded by combinations of a few more primitive commands. The cost is that some of the control and parameter passing operations that would have been executed within a command in the (compiled) implementation language of the program will happen instead under control of the macro executive. An assembled scheme, rather than the much slower command interpreter, is called for to minimize the command execution overhead due to the macro executive.

The low execution overhead (less than 200 μsec per command in this application) has additional advantages in a real-time system: several commands in a macro may perform an operation requiring on the order of one msec to complete. In an interpreted scheme, such an operation needs to be coded within a single command, and so can not be modified by users that do not have access to the program source code. With control of more of the time-critical functions of the instrument the user can customize real-time operations (such as control parameter updating and data transfer) for the specific application.

The software is designed for users with a broad range of requirements and programming proficiency that can be distinguished into four groups, each interacting with a different level of software. The casual user wishing to run established protocols invokes turn-key macros from a macro library. Different macro libraries are built to satisfy the requirements of different users (imagers or spectroscopists, for example). The macro nesting feature allows these libraries to be loaded on top of more basic macros that provide functions needed by all. Other users want to develop and test new experiments. They typically write new macros that include existing macros and command primitives. A few sophisticated users will require new commands to be written in the implementation language and installed into the program. Finally, fundamental changes in the system that involve modification of the control program executive and the low level code of the various instruments are the province of the applications programmer. A well defined interface exists for each level of interaction, facilitating simultaneous development by more than one programmer.

The program is designed for general purpose signal processing operations, such as data scaling, type converstion, FFT and matrix maniipulations; and for instrument control. In addition, the command set is configured for the specific applications of NMR imaging and spectroscopy. Both require facilities for developing and running pulse programs and interactive one dimensional graphics. Imaging experiments require additional capabilities : to generate complex waveforms, and to display and manipulate images. The data transfer rates between spectrometer and memory are much more rapid in imaging, typically on the order of 10,000 complex words (80,000 bytes) per second. The architecture of the spectrometer determines the maxmum permissible response time of the control program. In this application , the spectrometer consists of several instruments, each with its own control processor. The most rapid response required of the control program is about 1 msec; all faster control functions are handled by local processors in the instruments.

2 Target System

The instrument control and data analysis program was built to operate spectrometers designed at the Francis Bitter National Magnet Laboratory at MIT [1]. Each broadband (5 to 500 MHz) spectrometer consists of four instruments: an averager, RF electronics, a pulse programmer, and a waveform generator, as diagrammed in Figure 1. Each instrument has a local Z80 control processor and its own instruction set, and communicates with the host over a common IEEE 488 bus. The program supervises the processors and handles data transfer during an experiment in a synchronous fashion, with overall experiment timing controlled by the pulse programmer, with a timing resolution of 0.1 μsec. The host is a general purpose microcomputer, the Motorola 68000 based IBM S/9000 (IBM Corp., Armonk, NY), equipped with a SKYMNK array processor (Sky Computers, Inc., Lowell, MA). Graphics displays appear on the IBM S/9000 screen, while images are sent to a separate display unit (an IBM PC-AT with a 1024$\times$1024 pixel image display by Imagraph Corp., Woburn, MA). Host systems may have 2 to 8 Mbytes of memory, with the larger sizes required for imaging applications that are too rapid to allow data acquisition to disk. Resident code, intrinsic data structures and macro storage space requires about 1 Mbyte, leaving the remainder available for user allocated dynamic data structures. The code is written in Pascal and 68000 assembly language, and makes extensive use of multi-tasking, inter-task communication channels, and memory-mapped inter-task communication. The code is transportable to other computers with real-time operating systems that support such features.

3 Program Architecture and Implementation

Execution of a command or macro is controlled by an executive, which is also responsible for management of user defined variables. To respond quickly enough to control data acquisition from a high-level user defined macro, command primitives (henceforth commands) must be resident in memory when invoked. Commands are Pascal subroutines compiled and linked into the program, with their start addresses stored in a command table. Parameters are not passed to the commands using the intrinsic Pascal argument passing mechanism, but rather through a set of global argument pointer variables. When a command is invoked from the keyboard, it is located in the command table, its arguments are parsed by the executive, resolved, and the argument pointers are set. Execution then jumps to the start address of the command. Error detection is incorporated in each command subroutine, which may also write responses on the screen and prompt for missing arguments.

Macros consist of data structure declarations and a sequence of commands (including branching and looping instructions). They reside on disk as text files and may be edited from within the program. Our experience indicated that a simple interpretive batch command processor would not be fast enough for real-time control. At the time a macro is loaded into memory, all symbolic references to primitive commands, other macros, parameters, variables of local or global scope, and constants are resolved and stored in compact form. The macro storage data structure is a linked list of records, one record per instruction. When the user invokes the macro, the executive simply loads the argument pointers for the first command, jumps to the start address, and repeats the process on return until the macro is completed. This approach yields a command execution overhead (in a macro) of about 150 μsec. There is no limitation on macro nesting, but recursive calls are prohibited.

In the case of nested macros, the called macros must be loaded before the calling macro so that their start addresses in the macro table can be included in the executable code of the calling macro. This is not a practical restriction, but does prevent system initialization from a macro, because a macro cannot be both loaded (that is, assembled) and called by a another macro (because the calling macro could not be loaded). A simple batch utility is included for this purpose that executes commands and macros sequentially, just as if they were typed from the keyboard. Thus a batch file can load a set of macros and also execute some of them to put the spectrometer into a specified state. Batch files are useful for customizing the environment at program startup.

The program is currently implemented with two tasks: the main task and the acquisition task, as illustrated in Figure 2. The main and acquisition tasks are controlled by almost identical executives of the type described above. The main task receives its input from the keyboard, while the acquisition task receives input from an inter-task communtication channel (ITC) connecting it with the main task. When the string invoking a command or macro is typed from the keyboard, it is parsed and the macro and command tables are searched for the symbol corresponding to the instruction token. Main task tables are searched first. If the macro or command is not found the token is passed over the ITC to the acquisition task executive, and the acquisition task tables are searched. By this mechanism, main task macros may invoke macros and commands in both tasks, while acquisition task macros (executed exclusively by the acquisition task) may only contain references to commands and macros in the acquisition task tables. This asymmetric two task structure allows acquisition task macros to be run independently of the main task, in what is referred to as detached mode. While a detached acquisition macro is executing, the user may continue to execute main task commands and macros. The acquisition task runs at much higher priority than the main task, so that user activities on the main task during an experiment do not disrupt its timing. The main and acquisition tasks each have a window on the console display screen so that the user can conduct independent dialogs if desired. The most common application of this feature is for the acquisition task to post information on the progress of an experiment while data processing or other activities are performed by the main task. The tasks can be synchronized, if necessary, using global variables.

The run-time environment provided by the executive for each task includes error handling, debugging tools and a macro termination command. Errors detected within primitive commands result in messages and calls to the executive's error handling routine, with a parameter indicating the severity of the error. When the error handler is invoked, the macro line being executed is disassembled and printed on the console screen, along with the macro name, line number, and the symbols or values assigned to each argument. Depending on the parameter passed to the error handler, execution may resume or stop, jumping out of all nested macros in the latter case. The line number reported was assigned at load time, and may be compared with a listing file. The executive may be placed in a single step mode for debugging, wherein a message similar the the error handler's appears on the screen for each line executed, with stepping controlled by the return key. Finally, executing macros are stopped by typing Control-Break, which produces a query asking which task to stop.

User defined variables are stored in a variable table accessible to both tasks, with each entry having scope, type and protection fields. The scope field indicates that the variable belongs to either a loaded macro, the global scope (which may be created and modified from the keyboard or a macro), or the null scope (indicating that the record in the table in not assigned to any variable). Variables of the global scope are used for storing results that must be available after a macro has completed, as well as for setting flags between the main task and an executing acquisition task macro. For most purposes parameter passing is the preferred method for communicating among macros. The type field determines what values the variable can assume and the operations that can be performed on it. The requirement of type compatibility detects many programmer errors. A number of data types specific to the NMR application include: unsigned bytes for image display, and complex types consisting of pairs of 16 and 32 bit integers for data acquisition. Data types are listed in Table 1. Variables are entered in the table either by executing a command or during macro loading, where they are specified by variable declaration statements with syntax similar to Pascal. Parameters required by a macro are also declared at load time. When the macro is run the executive sets the entry in the variable table corresponding to each parameter to point to the specified argument. Variable storage and parameter passing are described in Figure 3.

A particularly inconvenient shortcoming of many spectrometer control programs is that variables internal to the commands are not accessible to user macros. Examples of such variables include shot counters, pulse lengths, the spectrometer frequency, cursor positions, and display scales. We have included a construct called the system variable, or SYSVAR, which is a global variable in the variable table pointing to a Pascal variable in the primitive command code. Such variables are generally assigned read-only protection so that the user can only modify such internals by executing the corresponding command, but can observe their state and use the variable in a macro.

In addition to scalar variables, the program supports dynamic allocation of array storage under user control. A special type of variable is used as a pointer, containing in its value field the address of a multidimensional array descriptor called a buffer control block, or BCB. The BCB specifies the data type and dimensional information of the array and contains the start address of a data space that is allocated when the buffer is created (Figure 4). Buffer size is limited only by available system memory. Six buffer data types accomodate fixed and floating point data. Buffers may be created and deleted at will, limited only by system memory size. The same data in memory can be simutaneously accessed under different buffer descriptors, as illustrated in Figure 5. This allows in-place type conversions, as well as customizing the data structure to match the operation. For example, a multi-echo image data array will generally be acquired as a two dimensional structure: the echo train from each excitation pulse sequence (and phase encoding gradient value) is stored following transfer from the averager to the host. For processing it is convenient to view the data as a three dimensional structure so that an image may be reconstructed for each echo. Equivalent buffers result in several pointers to a given data array which all have equal priority. The data space remains allocated until all pointers to it are deleted.

A file system allows storage of buffers on disk with the same multidimensional structures. As with buffers, the type and dimensions of files are specified when they are created. Read and write access to subspaces of the data file is supported, with the minimum transfer size being one row of the array. Files of text may also be read and written, and the contents of variables converted from the internal representation to ascii (and the reverse). This feature is most often used for header files that describe the state of the machine during a particular experiment. The state of the machine, including user defined variables, can be restored by reading such files.

4 The Command Set

The command set contains over 200 primitives. Some primitives exist in only the main or acquisition task, while some are duplicated as shown diagramatically in Figure 6. Scalar arithmetic, branching, operations involving creation of variables, buffers and files, and graphics display operations are all performed by the IBM S/9000 and are available in either task. Commands involving buffer arithmetic are implemented using the array processor and are contained exclusively in the main task. All commands requiring direct communication with the spectrometer instruments are installed in the acquisition task.

The basic instruction set shared by the tasks has a syntax similar to assembly language, i.e. an operation followed by several source and destination arguments. The instructions execute very rapidly with the simple macro assembler. The lack of an expression evaluator is not a great inconvenience for the instrument control and (mostly vector) data processing applications of this program.

An extensive set of commands for mathematical manipulation of buffers is implemented in the main task using the SKY array processor, supplemented with some assembly code for byte operations. Special functions for NMR and almost all the routines in the SKY subroutine library are installed as commands. A syntax was developed that uses wildcard characters to support vector operations with implied loops over multiple array dimensions. An illustrative example is

$$\text{BSMUL A AND BUF1} * * \text{ TO BUF2} * 1 * \ ,$$

wherein the two dimensional buffer BUF1 is multiplied by the scalar A and the result placed in a subspace of the three dimensional buffer BUF2. Here the deepest nested loop is specified by the leftmost $*$. This syntax frees the user from continuously typing range specifications, and facilitates writing general purpose macros which operate on buffers of arbitrary size. A similar syntax with wildcards that signify loop depth ($*1, *2, \ldots *5$) is used for file operations and buffer transposes. A proposed extension would permit operations on a part of a row or column of a data buffer, with the wild card replaced by special type of variable.

The primitives controlling data acquisition are divisible into two categories: those that communicate directly with the instrument Z-80 microprocessors and those that do not. All commands of the first type exist exclusively in the acquisition task and perform such functions as moving data from the averager; loading waveforms, synthesizer tables, and pulse programs, and starting experiments. These commands directly initiate communications over the IEEE 488 bus which need to be carefully synchronized to insure correct experimental results and rapid data throughput. Since the commands are in the acquisition task only, they are inaccessible from the keyboard or main task macros while a detached acquisition task macro is running so that unintentional interference with a running experiment is prevented. Commands that communicate with the instruments indirectly set experimental parameters from both the main and acquisition tasks. These parameters, for example pulse lengths, amplitudes and phases, need to be adjusted while detached acquisition macros are running. When commands of this type are executed, packets of update information are assembled and queued for transmission to the various instruments. A primitive of the first type (acquisition task only) causes these queues to be flushed. The timing of all IEEE 488 bus transactions is determined by the sequence of commands in a (detachable) acquisition task macro written by the user.

Synchronization of the acquisition task with a running pulse program is accomplished through commands of the first type. The acquisition task sends an instruction requiring a reply from the Z-80 when a condition is satisfied, such as 'tell me when recycle delay is over', followed by a request to read on the IEEE 488 bus. The acquisition task will then be suspended by the operating system until the read request is satisfied. If an illegal condition exists, such as issuing the above command after recycle delay when the next shot has already begun, an error condition is returned. Thus it is the burden of the acquisition task to keep up with the pulse program. Because the acquisition task runs at higher priority and task switching is rapid, main task activities do not interfere with acquisition activities. If the acquisition is very demanding the main task may receive few processor cycles and run very slowly.

5 The experiment development tools

The user wishing to develop new experiments must be able to write pulse programs, describe waveforms, and create macros for acquiring and processing data. The pulse program compiler and many of the commands for spectrometer control were derived from a program for spectroscopy designed by D. Ruben of MIT. Most modifications to this code, and parallel changes in the Z-80 instrument control code, were motivated by the speed and data transfer requirements of imaging.

The pulse program compiler is installed as a command in the acquisition task. A pulse program is created as a text file using the internal editor (or any external editor). The pulse program object code is automatically queued for downloading to the pulse programmer Z-80 following compilation. When an experiment is run, the Z-80 merges the object code with updated parameters for each shot and downloads it to the pulse programmer hardware for execution. This is how parameters are changed on the fly.

The waveform compiler allows complex waveforms to be expressed in a Fortran-like syntax. It is adapted from a compiler written by D. States for IBM. Because this compiler has an expression evaluator, waveforms are coded in a more compact and intuitive form than if they were specified directly in the macro language. Waveforms are downloaded directly after compilation to the waveform generator Z-80, which subsequently loads the values into FIFO's for D-A conversion (clocked by the pulse programmer) during experiment execution. The hardware incorporates waveform scaling and offset controls which are adjustable on the fly using aquisition primitives of the second category described above.

Frequency switching during multislice experiments is done in a manner similar to waveform generation. A buffer of table values is generated by a macro (taking the place of the waveform compiler) and downloaded to the spectrometer control Z-80 by an acquisition command of the first type. Synthesizer frequencies are switched under pulse programmer control.

The flexibility of high level acquisition macros is nicely illustrated in the case of moving data from the averager to a buffer in the host. For simple experiments with long recycle delays, typical of spectroscopy, a command to get the data can simply be issued during the recycle delay and the data sent immediately over the IEEE 488 bus. In order to optimize concurrent use of the Z-80's and IEEE 488 transfers, command primitives are also available to perform the transfer in two steps: move the data from the averager memory to the Z-80 memory, and subsequently move the data from Z-80 memory to host memory. The first operation can take place during a short recycle delay while the IEEE 488 bus is fully occupied with setting up the next shot, and the second step can be done during pulsing, which is typically quite long and without IEEE 488 traffic. The choice of one method or another only involves the high level commands in the acquisition macro.

As examples of the programming environment described herein, an acquisition macro and a processing macro are included in Figure 7,8.

6 Conclusion

The spectrometer control and data processing environment described above, and libraries of macros designed to support imaging and *in vivo* spectroscopy, are currently in use at the Lawrence Berkeley Laboratory, the Massachusetts Institute of Technology, and the Brigham and Women's Hospital. Systems ranging from a 4.7 T animal spectrometer to a 0.5 T whole body imager are supported by this software. Experiments which have been successfully implemented include multi-slice, multi-echo imaging, fast steady state free precession imaging, FLASH, ISIS, and ^{31}P spectroscopy. The flexibility of the programming structure allowed very rapid development of these diverse and specialized macro libraries.

This work demonstrates that instrumentation developed around standard hosts, buses and operating systems can yield research tools with performance comparable to very specialized systems. The combination of low macro instruction overhead, provision for access to system internals and a set of primitive commands for controlling basic acquisition and processing functions provides a foundation upon which libraries of macros may be built to serve a broad range of users, perhaps more effectively than a system with many specialized commands and a more limited batch processor. Well defined program interfaces for installing primitive commands, as well as the ability to modify instrument control code (host and Z-80) significantly broadens the range of experiments accessible to the researcher.

The program described herein runs on a single host in a multi-tasking environment, but can be easily extended to systems with two or more processors. The intertask communication channels would then utilize network software, a feature available under several commercially available operating systems. System performance could be improved with one processor running the main task and another running the acquisition task, with choice of processors and operating systems optimized for each task. One can readily imagine the main task running under a UNIX derivate with the acquision task running under a real time operating system, or integration of the main task with rule-based instrument control software.

References

1) Haberkorn, RA, Ruben DJ, McCue P, Barklay H, Thakkar A, Neuringer LJ, *Scientific Program, SMRM Third Annual Meeting*, p.292, 1984.

Name	Description
L	long integer
Y	byte
B	boolean
R	real
I	integer
C	character
S	string
X	complex floating point
M	complex long integer
J	complex integer
P	buffer pointer
F	data file pointer
T	text file pointer

Table 1: Data Types

Figure Captions

Figure 1: Schematic representation of the spectrometer hardware, data buses (medium and large arrows), and control signals (small arrows) of the target system. Each subsystem is installed in a separate Multibus crate, and communicates with the host over an IEEE 488 bus.

Figure 2: System resources are allocated to two tasks according to task function: the main task handles data processing, the keyboard and system utilities; the acquisition task controls the spectrometer. The shaded area shows resources shared between the main and acquisition tasks. Commands are transmitted among the tasks over an intertask communication channel (ITC), while data is passed through shared memory.

Figure 3: Each record reserved for data storage has six fields. The name and type fields are self explanatory. Variables may be assigned several levels of protection. Most variables allow read and write access, but system variables are generally read only. Each variable is either global or assigned to a particular macro. This is reflected in the scope field, which is used in macro assembly and for variable table cleanup. The address field may point to the value field in the same record (a) or to the value field in a different record (b). Parameters are passed to macros through this mechanism. The address field of the formal parameter is set equal to that of the actual parameter (in the calling statement) when the macro is called.

Figure 4: Buffers are addressed via a pointer variable. The value field of the pointer is set to the address of a buffer descriptor, or BCB. This record contains type, size and dimensional information describing the buffer, which may have up to five dimensions. The start address and skip factors are used to access the data, which is stored sequentially in

dynamically allocated memory.

Figure 5: This example illustrates the mechanism of buffer referencing for a hypothetical macro. A global buffer pointer is passed as a parameter to a macro, so the address field of the parameter is set to point to the data field of the global variable on entry. When a command in the macro allocates memory for the buffer, a BCB (BUF1) is created and the memory obtained from the heap. If another buffer, BUF2, is set equivalent to BUF1 during execution, another BCB with different structure or type pointing at the same data will result. This BCB must be removed before the end of macro execution. The data will remain accessible via the global pointer variable until it is deleted. When no pointers remain the memory is released.

Figure 6: The command set is divided among the main and acquisition tasks as shown, with commands duplicated in both tasks listed in the darker region.

Figure 7: The simple acquisition macro ZG (for zero and go) runs an experiment for a specified number of iterations, displaying the cumulative signal after each shot. The results are returned in a buffer passed to the macro through the buffer pointer SGDATA. The function of this macro is analogous to the GO command in some familiar spectrometer control languages. The definition section of a macro extends from the MACRO statement to the START statement. The PAR statement defines the buffer pointer to be passed by reference when the macro is run, and the VAR statement defines internal variables. BREAK QUIT causes a jump to the label QUIT when a CTRL-BREAK condition is detected. The CTRL-BREAK condition can be set by the error handler when an error is detected by the executive or within a command, or by pressing the key. TASK informs the macro loader that this is an acquisition task macro, and may be run detached following the DETACH command. Variables preceded by a % sign have global scope, and some

are SYSVARs. The loop defined by the LOOP label and the BEQ %AVDONE LOOP statement is repeated until the specified number of shots are acquired. Each iteration of the loop must be completed before the pulse programmer starts the subsequent shot, or a timing error is reported. The timing of this loop can be optimized for various types of experiments by selecting one of several strategies for data buffering, transfer, and (optional) display.

Figure 8: The macro SPECTRUM computes the power spectrum of the buffer passed by the buffer pointer FID. An equivalent buffer is created for type conversion and a new buffer is allocated to receive the spectrum.

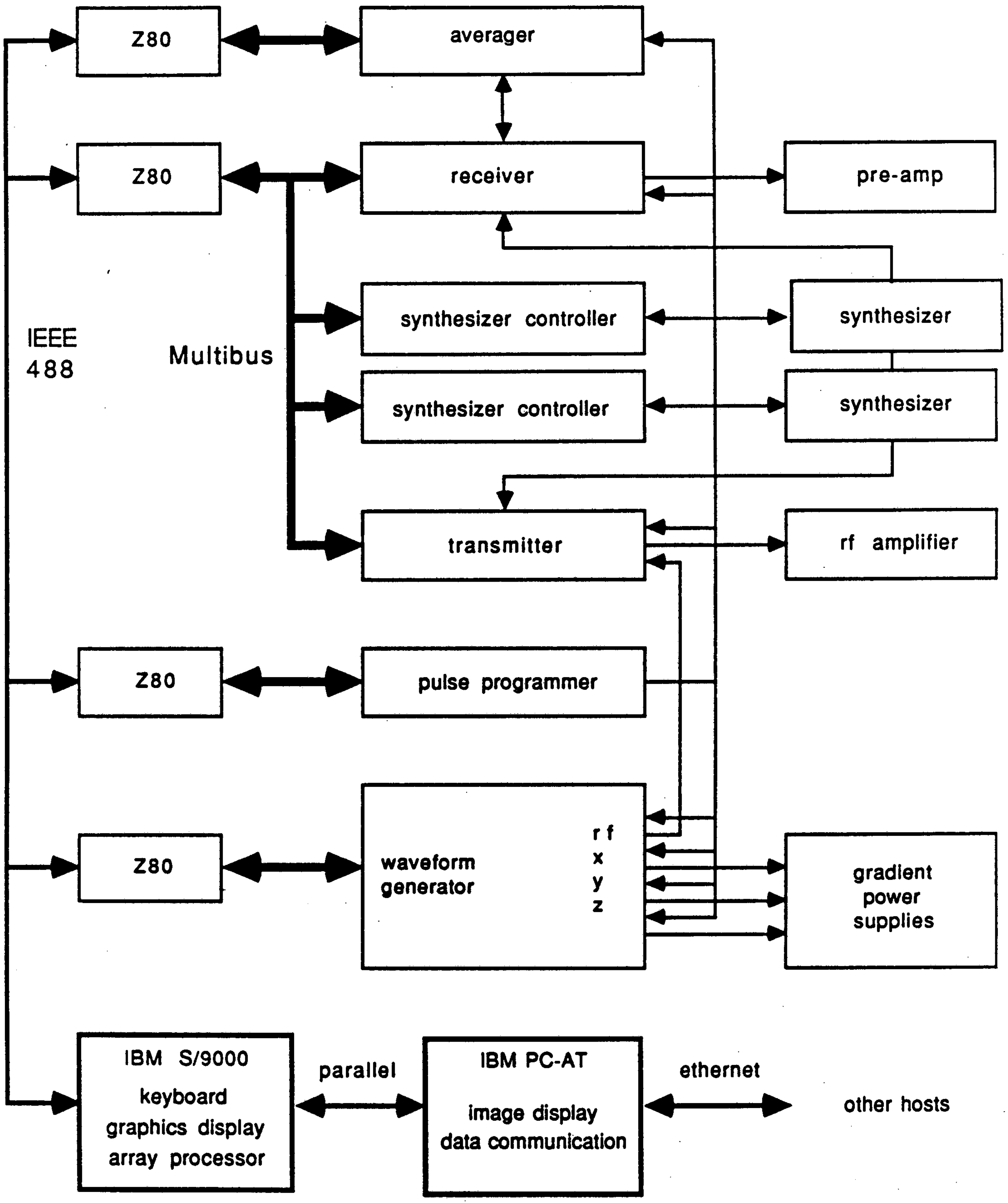

Z80
averager
Z80
receiver
pre-amp
synthesizer controller
synthesizer
IEEE 488
Multibus
synthesizer controller
synthesizer
transmitter
rf amplifier
Z80
pulse programmer
Z80
waveform generator
r f
x
y
z
gradient power supplies
IBM S/9000 keyboard graphics display array processor
parallel
IBM PC-AT image display data communication
ethernet
other hosts

Fig. 1

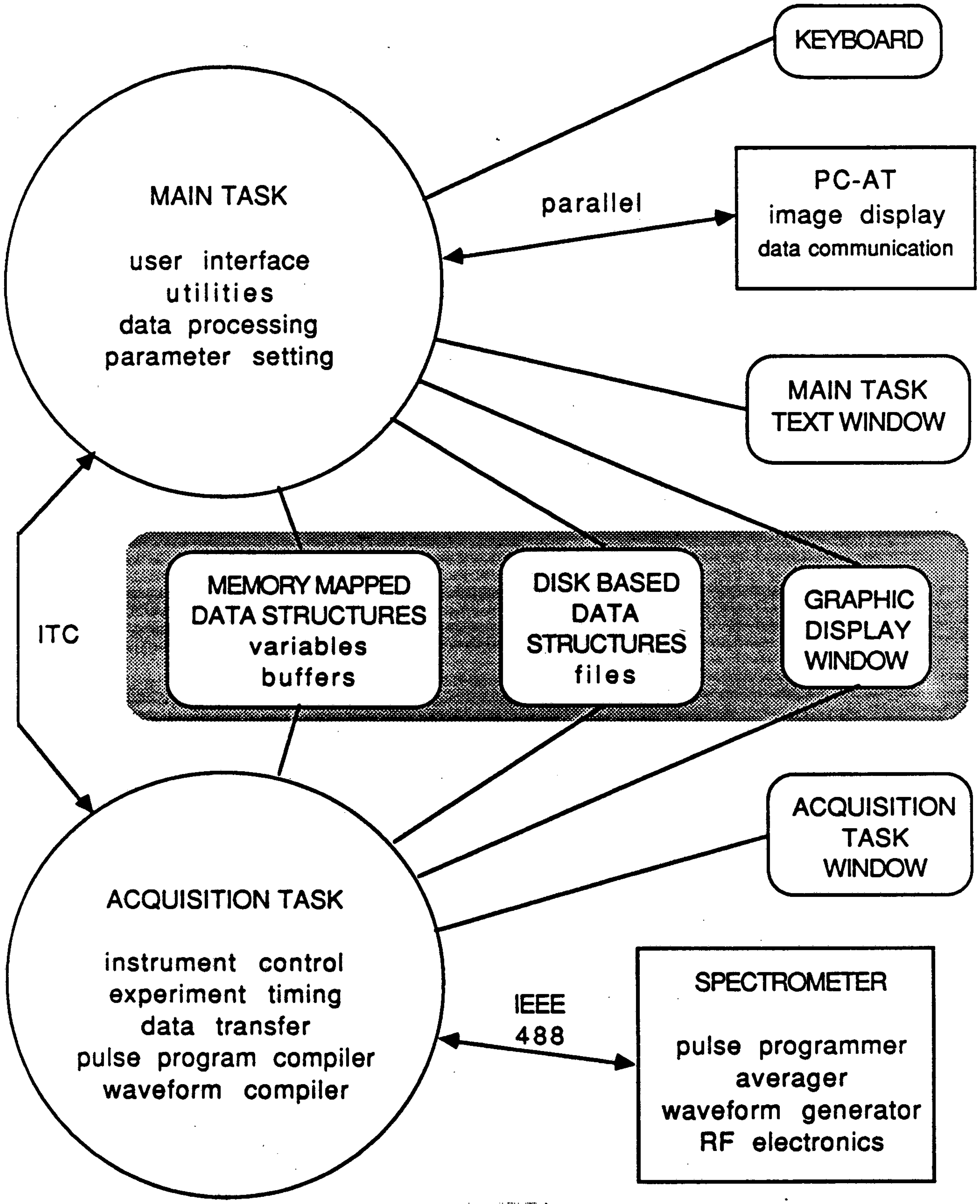
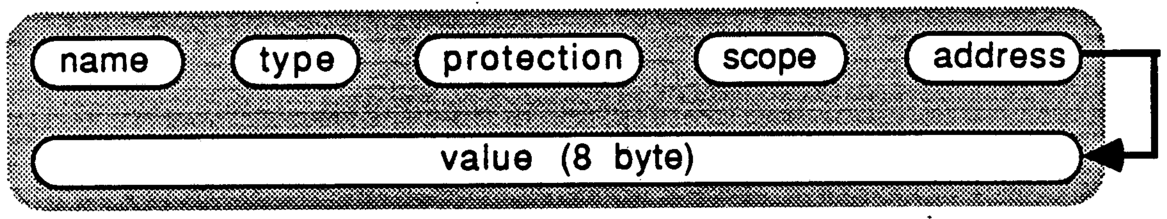
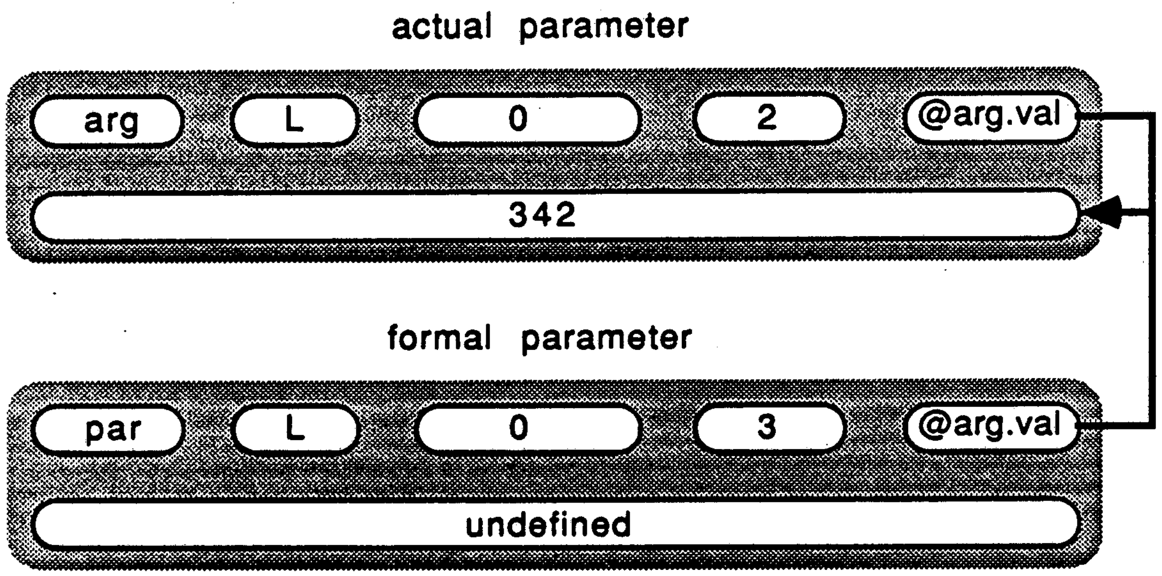

KEYBOARD
PC-AT
image display
data communication
parallel
MAIN TASK
user interface
utilities
data processing
parameter setting
MAIN TASK
TEXT WINDOW
ITC
MEMORY MAPPED
DATA STRUCTURES
variables
buffers
DISK BASED
DATA
STRUCTURES
files
GRAPHIC
DISPLAY
WINDOW
ACQUISITION
TASK
WINDOW
ACQUISITION TASK
instrument control
experiment timing
data transfer
pulse program compiler
waveform compiler
IEEE
488
SPECTROMETER
pulse programmer
averager
waveform generator
RF electronics

Fig. 2


name
type
protection
scope
address
value (8 byte)

(a)


actual parameter
arg
L
0
2
@arg.val
342
formal parameter
par
L
0
3
@arg.val
undefined

(b)

Fig. 3

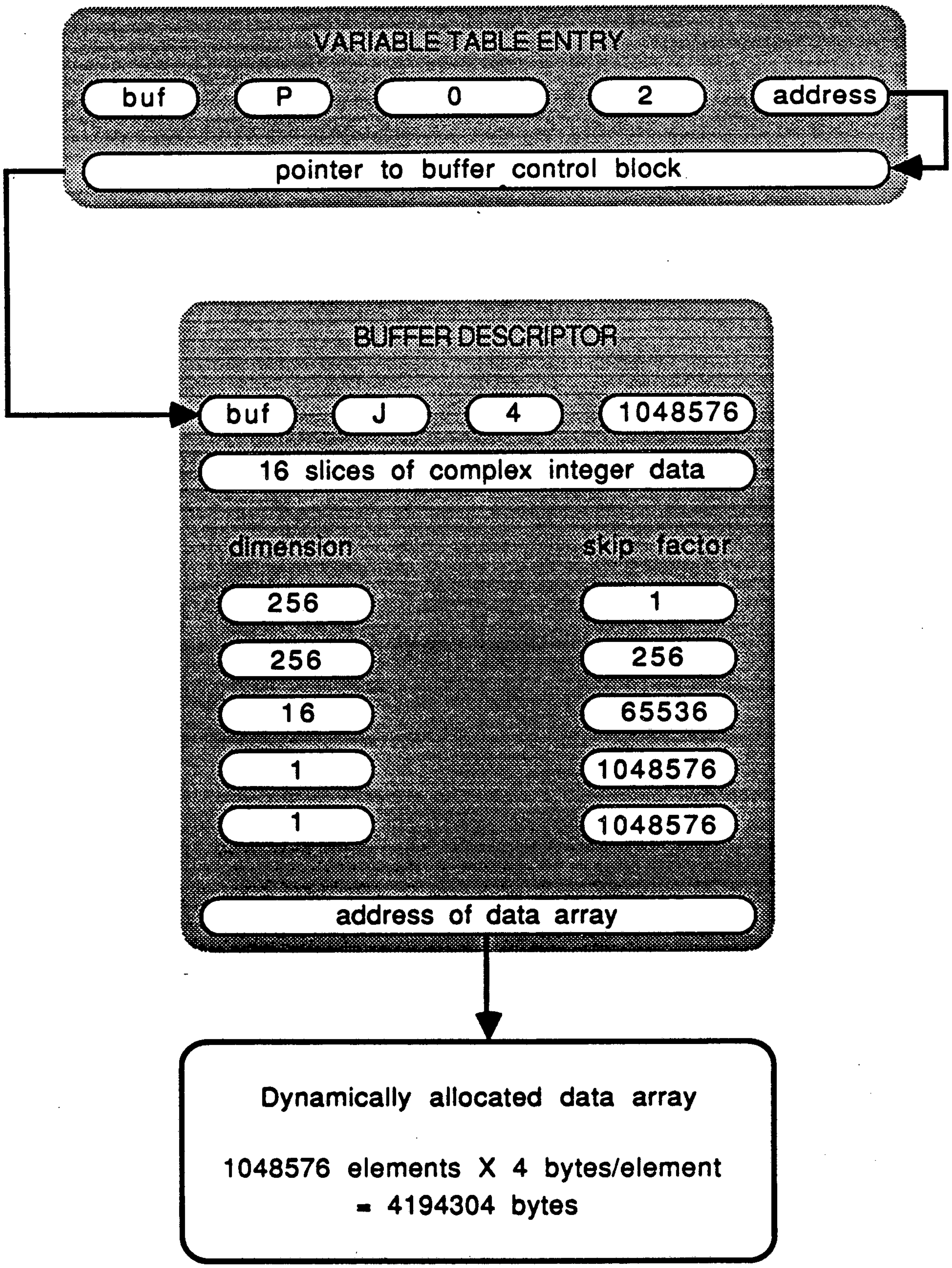

VARIABLE TABLE ENTRY
buf
P
0
2
address
pointer to buffer control block
BUFFER DESCRIPTOR
buf
J
4
1048576
16 slices of complex integer data
dimension
skip factor
256
1
256
256
16
65536
1
1048576
1
1048576
address of data array
Dynamically allocated data array
1048576 elements X 4 bytes/element
= 4194304 bytes

Fig. 4

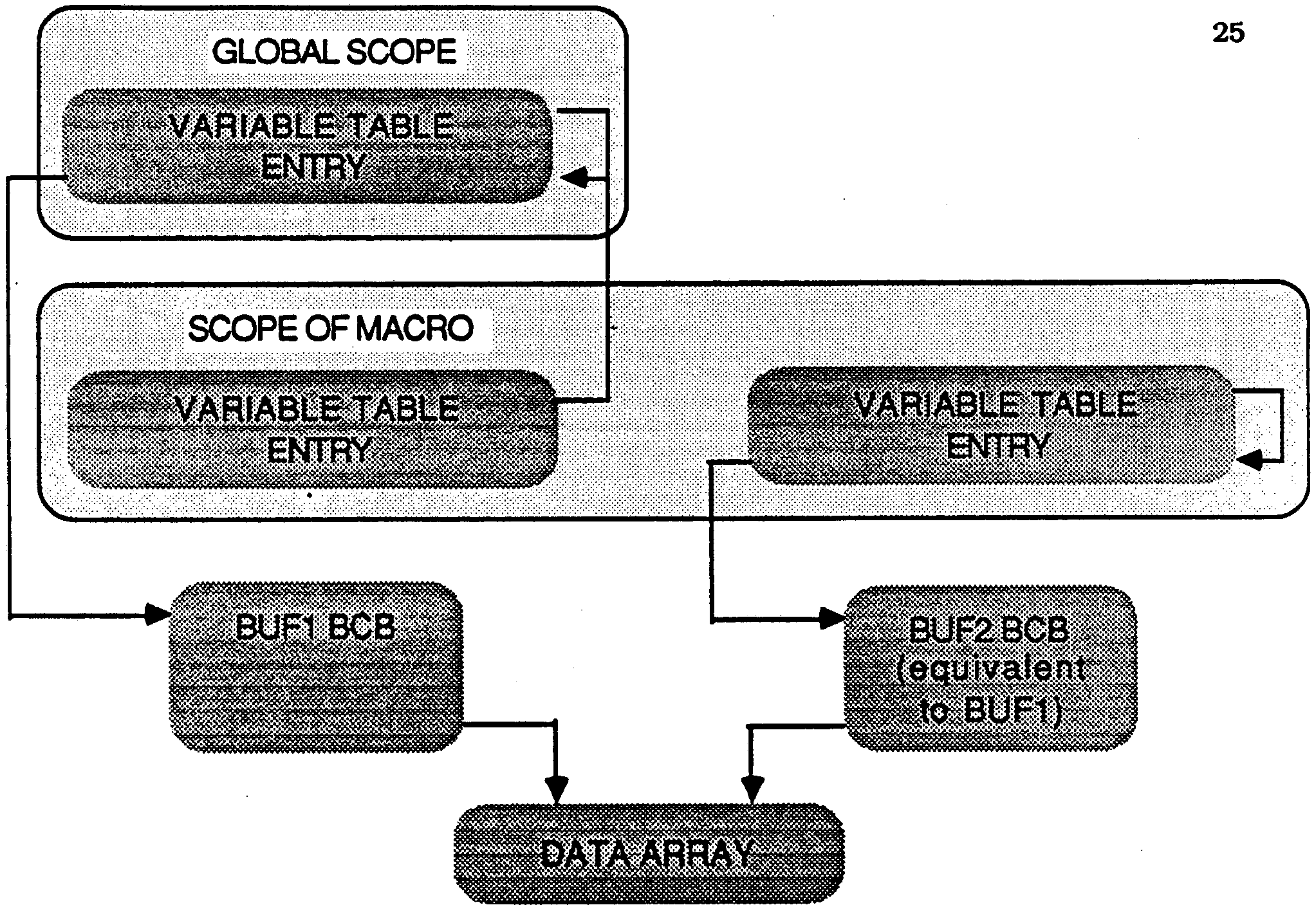

GLOBAL SCOPE
VARIABLE TABLE ENTRY
SCOPE OF MACRO
VARIABLE TABLE ENTRY
VARIABLE TABLE ENTRY
BUF1 BCB
BUF2 BCB (equivalent to BUF1)
DATA ARRAY

Fig. 5

MAIN TASK	editor operating system utilities (COPY, DIR,FREE) buffer operations using the array processsor parameter adjustment during experiments image display
MAIN AND ACQUISITION TASKS	variable definition scalar arithmetic string manipulation conditional branching buffer definition file system
ACQUISITION TASK	pulse program compiler waveform compiler instrument control averager data transfer experiment synchronization

Fig. 6

```
        MACRO ZG                     /cycle spectrometer and get data
                                     /arg1 = signal buffer pointer
        PAR SGDATA:P                 /declare formal parameter
        VAR DISBUF:P AVGSIZ:L        /declare local variables
        BREAK QUIT                   /jump to QUIT if CTRL-BREAK
        TASK                         /assemble as an acquisition task macro

        START                        /begin executable commands
        MUL %SIZE %NSAM AVGSIZE      /calculate size of data from system
                                     /variables %SSIZE and %NSAM
        ALLOC DISBUF 'M AVGSIZE      /allocate a buffer for display
        SET %PPQUIT %FALSE           /clear global quit flag
        DETACH                       /run independent of the main task

        PREP                         /initialize pulse programmer
.LOOP   RUNPP 1                      /run pulse programmer and put data
                                     /in averager block 1
        CHECKAV                      /verify data valid
        NEXTSHOT                     /prepare for next shot
        GAV 1 TO DISBUF *            /get data from spectrometer
        DISP DISBUF *                /display data
        BNE %PPQUIT QUIT             /check quit flag
        BNE %AVEDONE LOOP            /have enough shots been acquired?

        GAV 1 TO SGDATA *            /transfer data to passed buffer
        DISP SGDATA *                /display final result

.QUIT   SET %PPQUIT %FALSE           /reset quit flag
        BD DISBUF                    /delete buffer
```

Fig. 7

```
        MACRO SPECTRUM                     /process data to power spectrum
                                           /and display
                                           /arg1 = input buffer pointer

        PAR FID:P                          /declare formal parameter
        VAR XBUF:P POINTS:L BSIZE:L        /declare local variables
        VAR RBUF:P LBUF:P
        BREAK END                          /jump to END if CTRL-BREAK

        START                              /begin executable commands
        BDIM FID 1 BSIZE                   /get buffer dimension
        EQUIV FID * TO XBUF *              /make equivalent type 'X buffer
        BL2R FID * TO XBUF *               /convert to type 'X (in place)
        DIV BSIZE 8 POINTS                 /calculate size of intervals to be
                                           /used for baseline correction
        BBC POINTS AND POINTS AND XBUF * TO XBUF *
                                           /baseline correct
        BFT XBUF * TO XBUF *               /FFT (in place)
        ALLOC RBUF 'R BSIZE                /allocate type 'R buffer
        BMGSQ XBUF * TO BSIZE              /calculate magnitude squared
        BSQRT RBUF * TO RBUF *             /square root (in place)
        EQUIV RBUF TO LBUF 'L BSIZE        /make equivalent type 'L buffer
        BR2L RBUF * TO LBUF *              /convert to type 'L for display
        DISP LBUF *                        /display

.END    BD XBUF                            /delete local buffers
        BD RBUF
        BD LBUF
```

Fig. 8