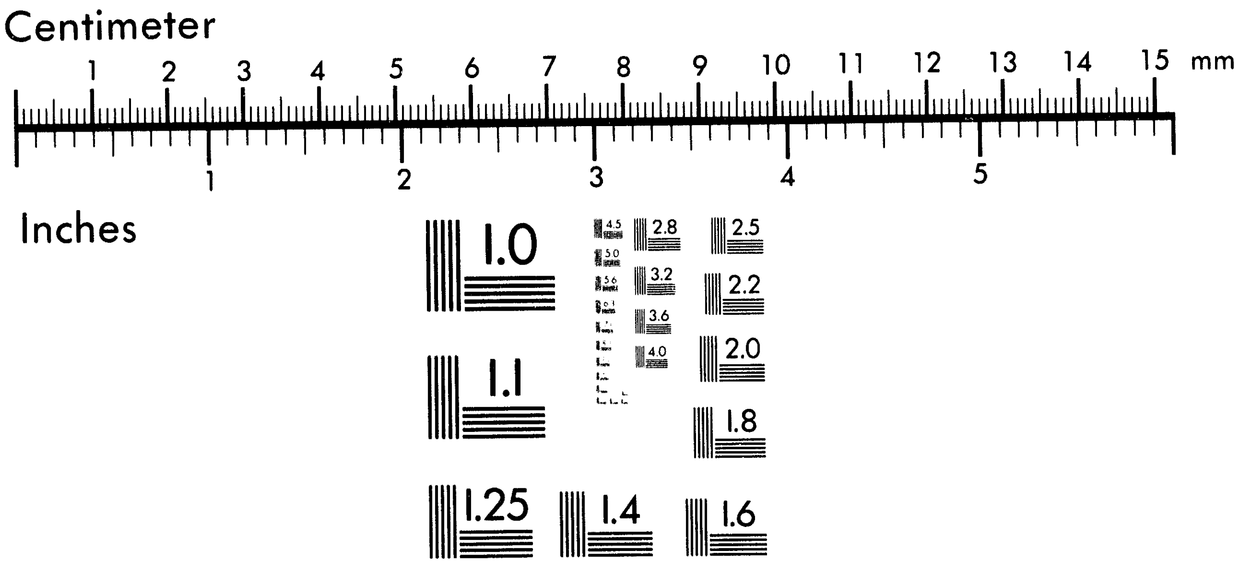
AIIM

Association for Information and Image Management

1100 Wayne Avenue, Suite 1100
Silver Spring, Maryland 20910

301/587-8202



MANUFACTURED TO AIIM STANDARDS
BY APPLIED IMAGE, INC.

Conf-940618--27 ANL/ASD/CP--81738

Development of GUS for Control Applications at the Advanced Photon Source*

Y. Chung, D. Barr, M. Borland, G. Decker, K. Kim†, and J. Kirchman
Argonne National Laboratory, Argonne, IL 60439, U.S.A.

Abstract

A script-based interpretive shell GUS (General Purpose Data Acquisition for Unix Shell) has been developed for application to the Advanced Photon Source (APS) control. The primary design objective of GUS is to provide a mechanism for efficient data flow among modularized objects called Data Access Modules (DAMs). GUS consists of four major components: user interface, kernel, built-in command module, and DAMs. It also incorporates the Unix shell to make use of the existing utility programs for file manipulation and data analysis. At this time, DAMs have been written for device access through EPICS (Experimental Physics and Industrial Control System), data I/O for SDDS (Self-Describing Data Set) files, matrix manipulation, graphics display, digital signal processing, and beam position feedback system control. The modular and object-oriented construction of GUS will facilitate addition of more DAMs with other functions in the future.

1. INTRODUCTION

The Advanced Photon Source (APS) is a dedicated synchrotron light source of the third generation being constructed at Argonne National Laboratory. As of this writing, construction of the accelerator systems is nearing completion and commissioning of the linear accelerators, positron accumulator ring (PAR), and the injector synchrotron has started recently.

Construction, commissioning, and operation of such complex machines as modern accelerators require a robust and versatile man-machine interface for effective control. As the size of machine and the number of subcomponents grow, the need for a flexible programming environment that can rapidly adapt to ever-changing situations is multiplied.

The control system for the APS is based on EPICS (Experimental Physics and Industrial Control System), which was co-developed by Argonne and Los Alamos National Laboratories.[1] The data link between the control points and the user applications is provided by Channel Access, the core of EPICS for communication between the host computers and the IOC (Input/Output Controller).

In order to complement the primary graphical user interface for EPICS and to handle streamlined data acquisition and analysis in a flexible laboratory measurement setting, a command line interface with both interactive and non-interactive programming capability has become necessary. Although most operating systems on various platforms provide a command interpreter and certain levels of programmability, they are not specifically designed for a laboratory environment dedicated to data acquisition, analysis, display and device control that requires command flow control, communication with measurement devices, and storage and archiving of acquired data.

Usually a dedicated, stand-alone application, though limited in its capabilities, would be developed for such purposes. The drawback of this approach, however, is that such applications provide little flexibility to adapt to different laboratory environments or different configurations of devices. This may become a serious shortcoming if frequent changes are required in the measurement procedures, e.g., during the construction and commissioning phases of the machine.

GUS is a script-based interpretive shell with command line interface that fills the gap between the operating system shell and stand-alone applications and thus overcomes the shortcomings of both. Users can allocate variables, scalars, and arrays, of numeric and string types. The data associated with these variables can be manipulated with the loaded set of commands and functions. One of the primary channels of moving the data between the external world and the GUS variables is the Data Access Modules (DAMs). The data thus stored in variables can be viewed, displayed, analyzed, and archived through the GUS commands and external applications.

The rest of this paper will describe the structure of GUS and its components, which comprise the kernel, built-in module, and Data Access Modules. How to interface to Unix for using modularized applications to expand the capability of GUS will be also explained.

2. STRUCTURE OF GUS

GUS consists of four major components: user interface, kernel, built-in command module, and Data Access Modules as shown in Fig. 1.

The GUS kernel is based on GPDAS,[2] which had been originally developed for the IBM PC and compatible microcomputers. GPDAS itself has been evolving ever since its first implementation on the APS magnet measurement facility but its capabilities were fundamentally limited by the memory restriction of the operating system. Unix-based machines generally do not have such restrictions and GUS has been significantly expanded in its capabilities, with the addition of several Data Access Modules, through modular construction.

The built-in command set provides the basic programming environment, and the DAMs add several more commands to it for data acquisition, manipulation, display, and archiving. GUS incorporates the native Unix operating system.

*Work supported by the U.S. Department of Energy, Office of Basic Energy Sciences, under Contract No. W-31-109-ENG-38.
†Present address: Samsung Heavy Industries Daeduk R&D Center, P.O. Box 43, Daeduk, Daejun 305-600, Korea

Commands that are not part of the loaded command set are passed to Unix for execution.

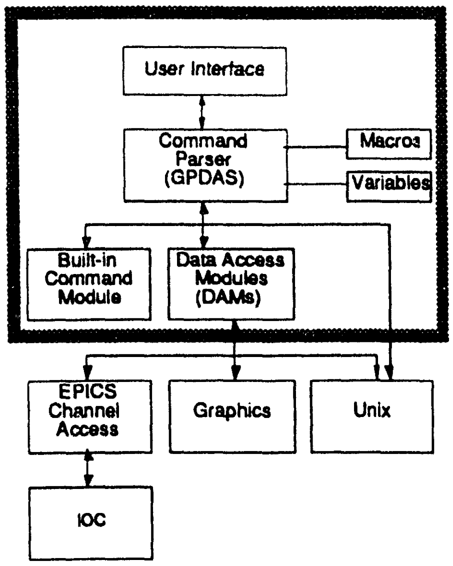


Fig. 1: Structure of GUS.

In consideration of the command interpreter's relatively slow execution speed, some routines requiring high speed can be separated as module programs and called from the shell or scripts. Modularization of an application into several small utilities is also a good programming strategy since they are easy to debug, don't take up much memory space, and can be used individually in other contexts. Such separation of functionality through small-size applications, however, needs optimization in terms of generality and efficiency. Smaller applications would be more general but also would tend to have less combined efficiency. Larger applications, on the other hand, would be more efficient at the expense of generality.

3. Kernel and Built-In Module

GUS commands, or statements, are first sent to the kernel for processing interactively through the console or non-interactively through a file called a script. In the interactive mode, individual yet complete statements are entered at prompts through the shell. After processing by the kernel, the commands will be executed by either the built-in module, a DAM, or an external program. A script is a collection of statements that is executed non-interactively and can be called from the shell or other scripts.

The typical command syntax of GUS is very much like plain English. For example, the following command

```
write line to screen from "Hello, World";
```

will print a character string `"Hello, World"` on the console. The semicolon (`;`) at the end of the statement is the command delimiter.

3.1 Data Types and Variables

The name of a GUS variable always starts with the #-sign and there are eight data types for variables: `char`, `short`, `int`, `long`, `float`, `double`, `string`, and `struct`. Arrays of arbitrary dimensions are allowed, except for the `struct` variables. The dimensions and size of the variable are limited only by the available memory.

Variable dimensions can be assigned either explicitly with variable declaration statements or implicitly by GUS commands through the context. For example, in the following statement

```
#y = sin(#x);
```

the dimensions of the `#y` variable will be changed, if necessary, to match those of the `#x` variable.

Declaration of variable types can also be implicit, e.g., by assigning a value to one without previous declaration. In this case, variable names that start with characters i through n or I through N implies `long` type. In contrast to the variable dimensions, which can change through the context at any time, variable types do not change once assigned unless explicitly redeclared.

3.2 Command Flow Control

The command control flow statements commonly found in programming languages are also available in GUS. These are `if`, `while`, `do .. while`, `for`, `goto`, `pause`, and `wait` statements.

3.3 Macros

Even though the plain English-style syntax of GUS statements provides better readability, it is sometimes desired to abbreviate them for conciseness. This is especially useful when a user is entering the commands interactively in the shell. An abbreviated way to write statements or parts thereof, called a macro, is available by using the `define` command.

Macros can contain multiple statements, including command flow control statements, and may have optional arguments within parentheses immediately following the macro name. For example, consider the macro definition:

```
define cagetvalue(pvs, data)
   chacc get value from pvs to &data;
```

"`Chacc`" is the Channel Access DAM command for reading and writing process variables (PVs). The following statement

```
cagetvalue(#bpm_names, #bpm_data);
```

would then get the data from all BPMs included in the name array `#bpm_names` and store them in the `double` variable `#bpm_data`.

3.4 Built-in Commands and Operators

In addition to the commands described so far, the built-in module includes commands for subroutine calls, file I/O, on-line help, save and show for variables and macros, binary data I/O for variables, script execution, and argument passing.

The operators provided by the kernel include arithmetic, string manipulation, array-related, mathematical, bitwise, data conversion, typecasting, file I/O, and other miscellaneous operators.

4. Data Access Modules

Data access modules are integrated into GUS as compiled objects. All of the DAMs at this time are written as a class in C++, which facilitates their addition to or deletion from the existing set of DAMs. For example, a DAM named "mydam" can be added with the following C++ statements in the program source code:

```
#include "mydam.h"
MyDAM mydam;
```

The C++ header file `mydam.h` has the class declaration for `MyDAM` and the DAM is integrated into GUS simply by instantiating a variable `mydam` of `MyDAM` class.

The constructor function for a DAM includes declaration of new commands for the DAM and the subcommands that belong to them. Each command or subcommand has an associated function to process the command issued by the user.

A typical command syntax for the DAMs is

```
command subcommand <arguments>
```

For example, the following command

```
matrix shift rows 1 9 by -1 in #x;
```

will shift rows 1 through 9 in a 2-D array variable `#x` by 1 row upward.

At this time there are eleven DAMs integrated into GUS as listed in Table 1. The `idlgraf` and `mateng` DAMs interface GUS to the commercial software IDL and Matlab, which make the features of these software programs available to GUS. Similar modules can be added in the future as necessary.

Table 1: List of Data Access Modules in GUS

Name	Description	No. of subcommands
chacc	Channel Access for EPICS	2
devio	Device control for EPICS	2
fdbk	Beam position feedback	1
file	File pointer navigation	4
idlgraf	Interface to IDL	7
mateng	Interface to Matlab engine	6
matfile	MAT-file read/write	7
matrix	2-D matrix manipulation	13
psgraf	PostScript graphics	9
sdds	SDDS file read/write	10
vector	Vector manipulation	13

5. Interface to Unix

Commands that do not belong to the GUS kernel or the DAMs are passed to Unix for execution. Using this feature, modular applications can be developed separate from GUS when high speed is necessary.

Data transfer and archiving can be facilitated through the Self-Describing Data Sets (SDDS) file format.[3] The data can be saved to an SDDS file through the SDDS DAM or other applications. The data can be either ASCII or binary, and users can generate one using a text editor such as emacs or vi.

An SDDS file is referred to as a "data set." Each data set consists of an ASCII header describing the data that is stored in the file, followed by zero or more "data tables." Each successive data table consists of a list of zero or more "parameter" values followed by zero or more rows of tabular data. The names, units, data types, and so forth of the parameters and of the columns of the tabular data are defined in the header. These do not change from one data table to the next.

Since the SDDS files carry detailed information about the data, they are useful for long-term storage of the data as well as for inter-application communication. A function library to support data I/O through SDDS files has been written and applications need to be linked with it in order to read and write SDDS files.

The transfer of data between GUS and other applications can also be done through `save` and `loadvar` commands as shown below:

```
save -g var to input.vm &#data1 &#data2;
myprogram input.vm output.vm;
loadvar all from output.vm &#data3;
```

In this example, variables `#data1` and `#data2` are saved to `input.vm` in binary format. The application `myprogram` reads them and the output is written to `output.vm`. The data can be retrieved using the `loadvar` command.

The applications used in this context are called GUS Expansion Modules (GEMs). The files that mediate the data transfer are called GEM-files and they contain the types, names, dimensions, and data of the variables. A function library has been written to support the data I/O. Due to the relatively simple file structure and I/O functions, the GEM-files are more suitable for short-term storage of data for communication among GUS and GEMs.

6. Applications

The earlier version on IBM PC and compatibles (GPDAS) continues to be used for magnet measurement and beam diagnostics R&D at the APS.[2] The current GUS runs on Unix workstations and is used to calibrate beam position monitors and test power supplies and is being implemented for machine commissioning and operation of the APS accelerator complex.

7. References

[1] W. McDowell et al., "Status and Design of the Advanced Photon Source Control System," *Proceeding of 1993 IEEE Particle Accelerator Conference, Washington, D.C.*, p. 1960, 1993

[2] Y. Chung and K. Kim, "Development and Application of General Purpose Data Acquisition Shell (GPDAS) at Advanced Photon Source," *Proceeding of 1991 IEEE Particle Accelerator Conference, San Francisco*, p. 1299, 1991

[3] M. Borland, "Application Programmer's Guide for SDDS Version 1," to be published.