International Journal of Engineering Inventions
e-ISSN: 2278-7461, p-ISSN: 2319-6491
Volume 2, Issue 6 (April 2013) PP: 48-56

Real-World Automation Control through the USB Interface

Robert H. Sturges

Abstract: *The shift in design of the personal computer to address applications in the area of multimedia and the obsolescence of ports for convenient access to hardware-software interfaces has begun to limit opportunities for the practice of custom-designed controls for automation. While several third-party hardware-software control packages exist, their learning curves and expense has not addressed the more fundamental needs of the designer to simply access and control a few bits or words of real-time information. This design shift has also given rise to the need for specialists in programming or information technology where the application may not support such effort. We address this need by providing a framework and examples of inexpensive hardware and simplified software with its logical design expressed in interdisciplinary form.*

Keywords: *hardware-software interface, automation control, logical design*

I. Introduction

The days of ubiquitous memory-addressable I/O are long gone (1950 through 1985), but the last vestiges disappeared only recently with the phasing out of the printer port, and the adoption of the "Universal Serial Bus", or USB. Thus, a large class of computer peripherals and related software were obsoleted long before they were functionally unable to compete. While printer ports existed one could very easily turn an external light on and off by simple programming. [1] In this paper, we will review the current needs to do this simple task. We will find that the newer protocols have added additional layers of software and firmware between the programmer and the real-world application.

Let us clarify the meaning of the term "real-world" in this instance. The intensely market-driven world of personal computers (PC's) has led to the seamless integration of multimedia with business support software and interfaces limited to keyboards, mice, and glowing rectangles. As this paper is drafted, we are seeing the replacement of keyboards and mice with touch-screen interfaces for the consumer who wants to manipulate pre-defined graphical images. Control or interactivity with anything else, such as a simple light bulb or non-standard keyboard is practically out of the question if it does not support a large consumer market segment. This "virtual world" by contrast, is pre-packaged and pre-defined for the consumer based on extensive analysis of taste and desires of a public conditioned to consider the computer as an appliance rather than a device for the implementation of specific logical needs. [2] The boundaries of "being digital" today are not surprising since the vast majority of installed computers are operationally dependent on a monopoly.

The notion of using USB for control of "real-world" devices, we will assert, is based on the relatively large capacity for computes at low cost compared with specialized control hardware. [3] Through our examples, we will touch upon the necessary investments today's peripheral provider must make to remain in the business

At the time of this writing, USB 2.0 is commonplace, while we await the appearance of the next level (USB 3.0) on the mass consumer market. Compared with the legacy methods of printer port and so-called "serial port" interfaces, USB offers advantages in speed, power, and convenience [4]. Disadvantages include lack of real-time response and the need to interface with unknowable codes written by others, and purposely not publicly documented. [5]

With respect to speed, USB 2.0 offers a "throughput" of 480 Mbps, but the timing of such data may be uncontrollable by the user. With respect to power, a built-in 500 mA source is specified at each "port", of which there may be up to 127 such ports. Thus, some low-power peripherals may no longer require the otherwise ubiquitous "AC adapter" or wall-wart in order to function. With respect to convenience, the USB specification requires "plug and play" interconnectivity. This means, in brief, that the user may not be required to suspend the operation of other running programs while the lower-level of connectivity is negotiated by the Operating System (OS) and the peripheral. Of course, this implies that the peripheral is now required to support an intricate series of handshaking activities not required previously. Fortunately, this software task has been implemented in firmware by several vendors of "interface chips", e.g. Future Technology Devices International, a/k/a FTDI. [6] On the minus side, while the USB standard is public knowledge, it spans several thousand pages of highly detailed technical data that differs in detail for each hardware implementation. [7]

II. Objective

Our objective in this paper is to enable educators, researchers, low-volume practitioners, and unique system creators to use the power of multi-GHz computing for custom programming not deemed relevant by the forces of market-driven "applications". We would note also that the need for "real-time" control still exists in laboratories. By this we refer to the practice of knowing precisely (to the microsecond) when a signal will appear at an interface for use by either party. Such real-time operations are only approximated by the alternative "synchronous service" of USB in the delivery of digital video data, but this ability is being phased out as the digital television market is supplanted by streaming technologies that do not need to support real-time transmission.

With the demise of analog television, a large market segment has also been eliminated from consideration, but real-time computing is still supported by a small cadre of laboratory-focused providers. [8]

Another key objective is to educate our general engineering student population, excepting perhaps those in pursuit of degrees in Computer Science, to the means needed to achieve "software literacy" consistent with industrial automation classroom activities. We find that no market-driven "applications" address this need.

2.1 Brief Literature Review

We acknowledge here the popularity of several alternative technologies directed to the control of real-world devices, for example, valves, relays, motors, indicators, etc. The programmable logic controller (PLC) has been in use for decades with offline programming methods based on then-current PCs. These devices typically employ a reduced-capacity processor for on-line implementation, performing sequential logical calculations in the same vein as parallel connected devices. In general these systems do not employ a run-time graphical user interface (GUI), and function as repetitive, customized controllers for a broad range of equipment systems with relatively low data processing tasks, e.g., RS Logix. [9] With such systems, an investment needs to be made in the acquisition and practice of proprietary computer architectures and languages.

For about 25 years, a unique software product from National Instruments [10] has enabled the creation of virtual instruments on a PC using a proprietary graphical "language". Teamed with that company's interface hardware, individual digital and analog I/O is made possible along with extensive logical internal processing. With care, one can program with procedural code embedded in the graphical media. For specialized applications, one can avoid the PC altogether and operate such virtual instruments using field-programmable gate arrays. Similar to the PLC's, an investment needs to be made in the acquisition and practice of proprietary computer architectures and languages, in addition to the I/O hardware. Recently, USB-compatible hardware for introductory use has been offered. [e.g., NI's "6009" I/O block product]

We have surveyed the contemporary marketplace for introductory volumes to support the available hardware and find that a relatively high level of programming skill is assumed, while the level of hardware knowledge assumed is quite low. This indicates a trend away from interconnection with the real-world in favor of the virtual world isolated from the exigencies of hardware save the screen and keyboard. This prospect and the market pressures that created it are well-addressed in research devoted to rapid obsolescence. [11]

III. Approach

Our approach to providing a kernel of knowledge which may serve the related needs of educators and the general engineering marketplace proceeds with the identification of a minimal set of interface hardware with which to establish the USB communication protocols at the "outboard" end of the information chain. (Figure 1) We then proceed to develop the "inboard" end of the information chain with the express interest in an "open" architecture for personal program realization. In both steps, we assume the minimum amount of specialized programming and electronic knowledge of the user.

To communicate (rather than encapsulate) this knowledge, we express here the basic functions that need to be performed irrespective of the hardware or software platforms selected. Only to express examples of the method do we resort to actual coding and select a hardware platform. The functional approach was developed decades ago by teams of "value engineers" intent on reducing the cost of military acquisitions by expressing the instance of a product or process by its generic functions, all logically connected by "and" and "or" links. [12] An example function block diagram is shown in Figure 2, wherein the highest level function is placed to the left, and its decomposed lower-level functions appear more rightward. Notice that the "flow" of the knowledge is expressed by answering the question "why" by looking towards the left, and the question "how" by looking towards the right.

The functions themselves are expressed by a single active verb and a measureable generic noun in a small rectangle. For example, "support load" is a viable function, whereas "provide support" is not. [13] There is no "right or wrong" function diagram, only viable and non-viable ones as judged by the team of users that employ them for interdisciplinary communication purposes. Please notice also that such a diagram is not intended to express the same information as a "flow chart" since the latter rarely explains the *raison d'art* for the functions and their logical relationships. In our example, the right-most series of functions, when read from the top to bottom concisely express the "pseudo code" prevalent in other articles. Taken all together, the function diagram expresses the intent of the designer or user.

In any such endeavor, one must chose software and hardware platforms to realize this intent. We have chosen the C-language for our software because is appears to be the most popular and most enduring choice. We will additionally assume a low level of expertise in C by the reader so that the essence of the project can be readily apprehended, but we will not be offering here a review of the elements of that language. We recognize the strong following of other languages such as C#, Java, and C++, but have avoided the object-oriented approach as not vital to the task at hand, and have steered away from proprietary implementations as much as possible. We have also chosen the Phidgets™ hardware platform since, for the present at least, the hardware and software implementation is more than adequate for the inexperienced user. We are in no case affiliated with that company, and several others, such as J-Works™ or Active Wire™ may serve as well.

Another choice needed to be made in this endeavor: that of the PC itself. Since the market is led by said monopoly, we have selected Windows™ as our OS, but you will find that others support the use we make of it equally well. The PC hardware choice is fully generic, so long as it has a USB-compatible port.

To recapitulate, the goals of this project were to illustrate by practical example the steps one must take to realize communication between an application of custom-developed code as practiced by a general engineering audience with equally customized hardware. The examples we will now discuss in detail comprise a general-purpose analog/digital interface ("ifKit"), and a commonplace R/C servo interface ("servo").

IV. Details of Implementation

The implementation of communication to a USB device in these cases comprises two separate instances. In the first instance, the hardware supplier has provided software that interacts with the user through a specialized window. This window displays the measured conditions of the external board at the "outside" end of the USB chain. The user cannot access the code from which this window appears, but can determine through its use the fact that the board may be attached and operating correctly. One may also "test" the outputs in some cases by setting certain fields within the window.

In the second instance, the programmer uses software calling sequences to communicate with the external board. These sequences are pre-compiled for the most part, so that again, the programmer cannot access the code through which this communication occurs.

Prior to the instantiation of either of the two instances, the user must install a set of pre-compiled codes called "drivers" into the OS of the target computer, in much the same way as "printer drivers" are installed for that purpose. Because one cannot access this code, the possibility does not exist for the user to specify the precise timing of any of the communication actions that take place. By dint of the high speed of the processor, one must assume a "level of service" that may be commensurate with the task to be performed, but there are no guarantees. Figure 3 illustrates the functionality of this driver (as much as can be known) by the isolated function block in the upper left-hand corner, and the software agent by the oval entitled "Phidget Software". These two fragments are connected by a solid line representing the functional "and" operator. In this example, the board is a Phidget type 1018 Interface Kit.

The instructions needed to install the drivers and to access the test window are given in the company's Programming Manual that was available on "the web" through that company's web-site, "Phidgets". (It is important to note that the rapidly changing nature of the hardware and software market requires the user to adopt a "digital" outlook and accept the fact that information may appear, change and even evaporate without notice at any time. In this paper we have avoided giving such ephemeral references wherever possible.)

Our purpose is to educate users and implement examples of the code that we do have control over, so our basic function is given as "Test Protocol". An equally valid basic function could be "Educate User" if the working team agrees. How to perform this basic function is specified by three sub-functions: Verify Communications, Display Changes, and Stop Communications. Each of these functions is further sub-divided as shown.

Taking first the Verify Comm's function, we need to "open" the board for communications, and inform the user (or his/her program) that we have done so. These two sub functions will comprise the most novel of the tasks for the programmer with respect to the hardware. The order (top to bottom) of each sub-function for these is arbitrary, but we have arranged the function block diagram (FBD) so that it reads at the fourth level like a flow chart vertically insofar as possible. As with any program in C, one typically declares the variables and their types at the beginning of the code. A new feature is next introduced, that of declaring a "handle" for the external device to be invoked by the code in the PC. This handle takes the place of the memory-mapped I/O of earlier days, and it is simply a pointer to an address that acts as the gateway for all communications with the device. A separate step is needed to create the space in memory for this handle. The detailed instructions for these steps are given in the on-line data base of the Phidgets web-site, or in the appendix which shows the actual C-code used by the authors.

To account for the possibility that the device is not physically connected to the USB port of the PC, a function is created that tests and waits for the attachment of the same for a user-specified period of time. During this time, the user may post a warning on the screen of the PC. After the negotiation for communication is made by the firmware of the device and the PC, the user may call a pre-written function to "open" the board for communication (reminiscent of the "open" command in BASIC). Each of these pre-written functions are known as the Application Programmer's Interface (API), and are supplied by the maker of the interface board so that communication with the same is facilitated. In the C language, these may be placed in so-called "header files". In this case, the major header file is named "Phidget21.h". Without it being present, the compiler will not successfully bring these functions into the user's code. Also, there will be a corresponding library of pre-compiled functions that must be linked to the compilation process. In this case the "Phidget.lib" file needs to be attached to the linker through its input dependencies listing.

As Figure 3 illustrates, there are functions needed to inform the user (or the user's program) of other activity: in the first sub-function, we invoke the API to fetch and display the serial number of the attached device and its physical and functional characteristics. This step is crucial if more than one of the same type of device is attached to the USB. The serial number found would be used to distinguish between the attached devices for steering the desired I/O. Each handle created would be associated with a separate serial number. In the second sub-function, error conditions are made visible, that is, an API call is made to extract details of the error condition. Also, a "call-back" function is defined to catch any errors that are thrown by the compiled code. This process involves the declaration of a function that will be invoked whenever an error condition arises. This function is of our own design but must be identified by the type "__stdcall" so that the compiler can establish a link between the internal error traps and the user's code. (There are *two* underscores before the letters "stdcall".)

The center portion of Figure 3 illustrates many opportunities to invoke the API provided by the Phidget company. Here, the programmer sets bits in an output register according to the numbers returned by calling analog input channels. In this case, a popular distance measurement unit (a Sharp 2Y0A02) returns an analog voltage in response to detecting a reflective object within its range. The setting of the output bits is illustrated in the C code with concern for reader transparency of the functions, not efficiency of the code itself.

Finally, the program is shown responding to a 'q' from the keyboard, indicating the end of the test. At this point, the API is again invoked with functions to close the communications and delete the memory space allocated in the handle creation step. Failure to do this may result in a build-up of inactive code, otherwise known as a "memory leak" as the host system begins to respond more slowly with repeated usage.

The entire C code is given in the Appendix of this paper for reference, but may be available on our website. <http://www.tbd...> In its recommended form, all comments are colored green, quoted text is colored red, keywords are blue, and the code body is in black ink. Line numbers are shown in cyan. Except for the "__stdcall" and a few necessary pointers, the C code is rendered as simply as possible for the inexperienced reader.

4.1 A Further Example

To illustrate more fully the programming conventions needed to bridge the "USB gap" for the educator and user of real-world devices, a second example will be briefly described. In this instance a USB-driven radio-control type (R/C) servo board is chosen to illustrate slightly more complex communications. It is a Phidget type 1061 Advanced Servo board. Figure 4 shows the FBD of the interface program, and one will note how similar it is to the entirely different hardware and software system represented in Figure 3. One of the major benefits of the FBD approach is to illustrate the commonality inherent in many artifacts that appear to be otherwise quite distinct. For this reason we will examine this example of the servo control from the viewpoint of the code itself, as it appears in the appendix.

The program begins (for the sake of the compiler) with the "includes". (It begins for us with the "main()" function located near the physical end.) We add <stdlib.h> since we will be using the "atof" function defined there. We continue with our global variables: a character to be read from the keyboard, a true/false indicator for the main loop, and an array of values to hold the target movements of each servo. We define eight of these, but will be using only two.

The first function is our "call-back" which is typed __stdcall. We name it ErrorHandler, and the parameter list comprises the device handle (SRV), a utility pointer that we will not be using, the ErrorCode and a pointer to the character string that describes the error itself. This function gets called by the compiled code whenever the system of Phidget API software detects a problem. Otherwise it is inactive. Like all int functions, we return a value (0) to the OS.

Our next function, display_properties, needs one parameter of type "CPhidgetAdvancedServoHandle". The C language encourages customized typing, and in this case makes it clear that the language is C, the vendor is Phidget, the type of interface board is AdvancedServo, and that the parameter itself is the pointer/handle to the gateway that transmits all information about the board to the program which uses it. Our local variables include facts about the board (to be determined) and a character string pointer for storing the name of the board. We then make a call to the API function to return the type of device connected. Its parameter list contains a seldom-used feature: the handle is "cast" as a "CPhidgetHandle" type just to be sure that the compiler knows to look in its ...phidget21 library. A list of this information serves as the initial data base for identifying the device, should there be multiple copies of the same type of board connected to the USB.

Our next function, move_servo, needs the familiar handle, SRV. This small function reads the keyboard to get a floating-point number by way of a character string of up to 10 characters (value). After scanning the keyboard, this string is converted to a double and saved in an array for use in the next line. We then call another API function which needs the handle, the index value of the servo motor to move, and the exact angle in degrees we command it to move to. Incidentally, the list of all the API functions available does not appear in any reference book, but rather "on-line" at the Phidget web-site. The interested reader must navigate to this list and its recommended usages by selecting the specific Phidget device from a list (in the left-hand column). The list is only a few steps below the home page, so we will not describe it in detail here.

The "big" function appears next, and it would be considered the "main" function but for the stylistic approach to C which keeps main mostly empty. Line 49 is both a declaration and an assignment, making space for the device handle. In older PC's this would be a fixed location in memory to which we would make all references to our attached device. The next line invokes an API function to actually reserve space for all the data associated with the handle and our attached device. We then call an API function to define the name of the error handler we defined earlier with our __stdcall. Now, the compiled code will compute the correct reference to invoke when an error is thrown. Finally, we call another API function to open the board for communications. The "-1" parameter is a requirement of the API.

We now ask if the device is indeed connected via USB (has the interface been negotiated satisfactorily?), and wait for a prescribed number of milliseconds if not. If no problems occurred, we collect and display the properties of the connected device. A for-loop follows which sets the properties for each servo dynamic response, and chose at this point to actually engage the servo into action. This last step can be inserted anywhere appropriate for our needs.

A user-terminated while-loop appears next, prompting the user to enter an 'm' to move a servo, or a 'q' to quit. The switch statement uses the ASCII equivalent of these characters. If we do elect to quit, we invoke a pair of API calls to close the board from communication and delete the memory allocated for it. The "main" that follows is the actual entry point for a C program.

V. Results

The practical result of this project is the implementation of a simple, low-cost demonstration board for the educator and first-time user of customized USB devices. Figure 5 depicts a 5 x 8 inch platform upon which has been attached the two USB interface boards mentioned above. Also included are a pair of infra-red (IR) distance measuring units and a pair of R/C servo units. The InterfaceKit 1018 also has a pair of LED's attached to two of its eight digital output ports. Not shown are the USB cables that attach to the host PC and the small power supply needed to power the 1061 board servos.

The intellectual result of this project is the understanding of the user-designed and written software which works through the "encapsulated" codes of the maker of the boards and the authors of an OS and C compiler for a PC. We have shown how to establish communication with USB-powered interface boards of differing functions with a few fundamental programming ideas. We also illustrate the use of a function-defined structure for designing the code at a high level before committing to writing at the level of the compiler. In this way, the sections can be examined by all stakeholders and agreement reached before the debugging process begins.

VI. Conclusion

This article has endeavored to provide implementation transparency for the developer and user of low-quantity, often unique interfaces without the need for specialists in "information technology" and computer networking. We look forward to the application of this work to industrial automation classroom activities, at least, and to implementations within the research communities far removed from "computer science". We have found that one hardware and software system worked very well, and could work across many platforms, but caution that many others exist and that we make no recommendations or endorsements of a particular vendors product.

We also recommend carefully exploring the API provided by the maker before choosing a USB interface board for your own purposes. Your choice of language may not be C, but the API you select must be compilable from your language of choice.

APPENDIX: Annotated Software for the Interface Kit and the Servo

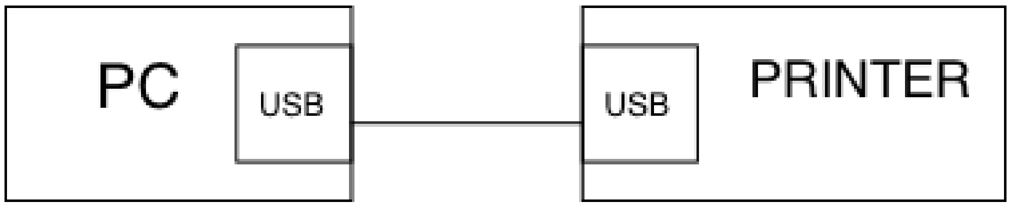

Figure 1. A User's High Level View of the USB Information Chain

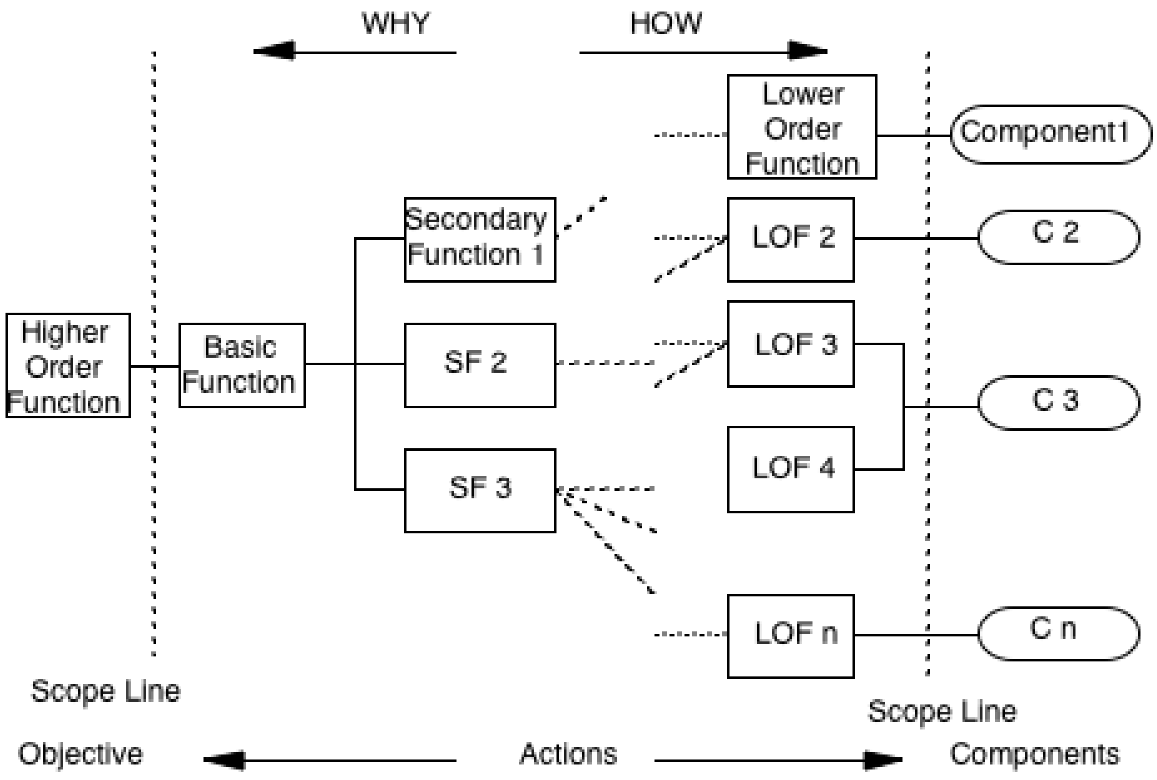

Figure 2. A Generic Function Diagram (After [14])

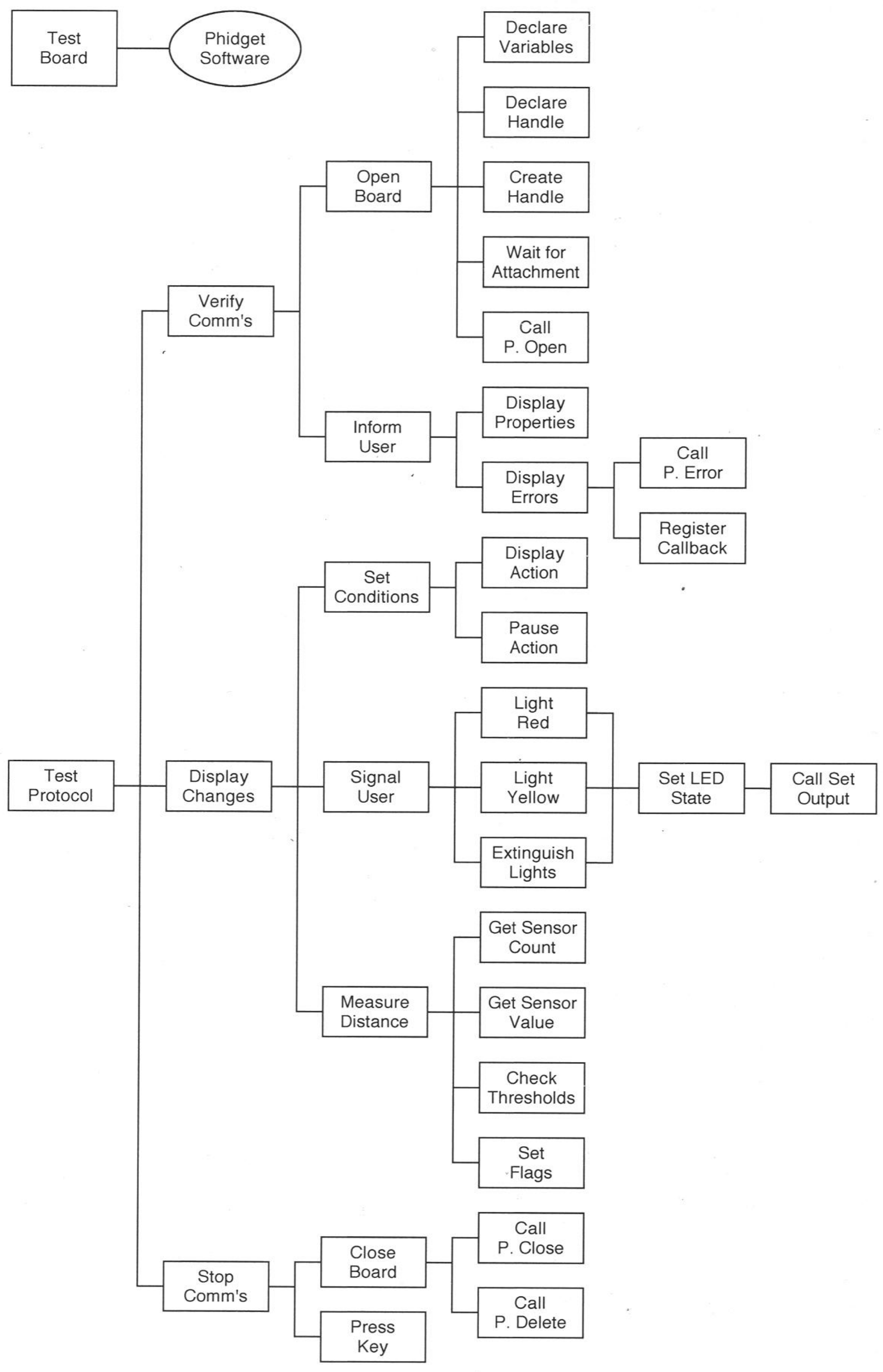

Figure 3. Function Diagram for the Interface Kit Software

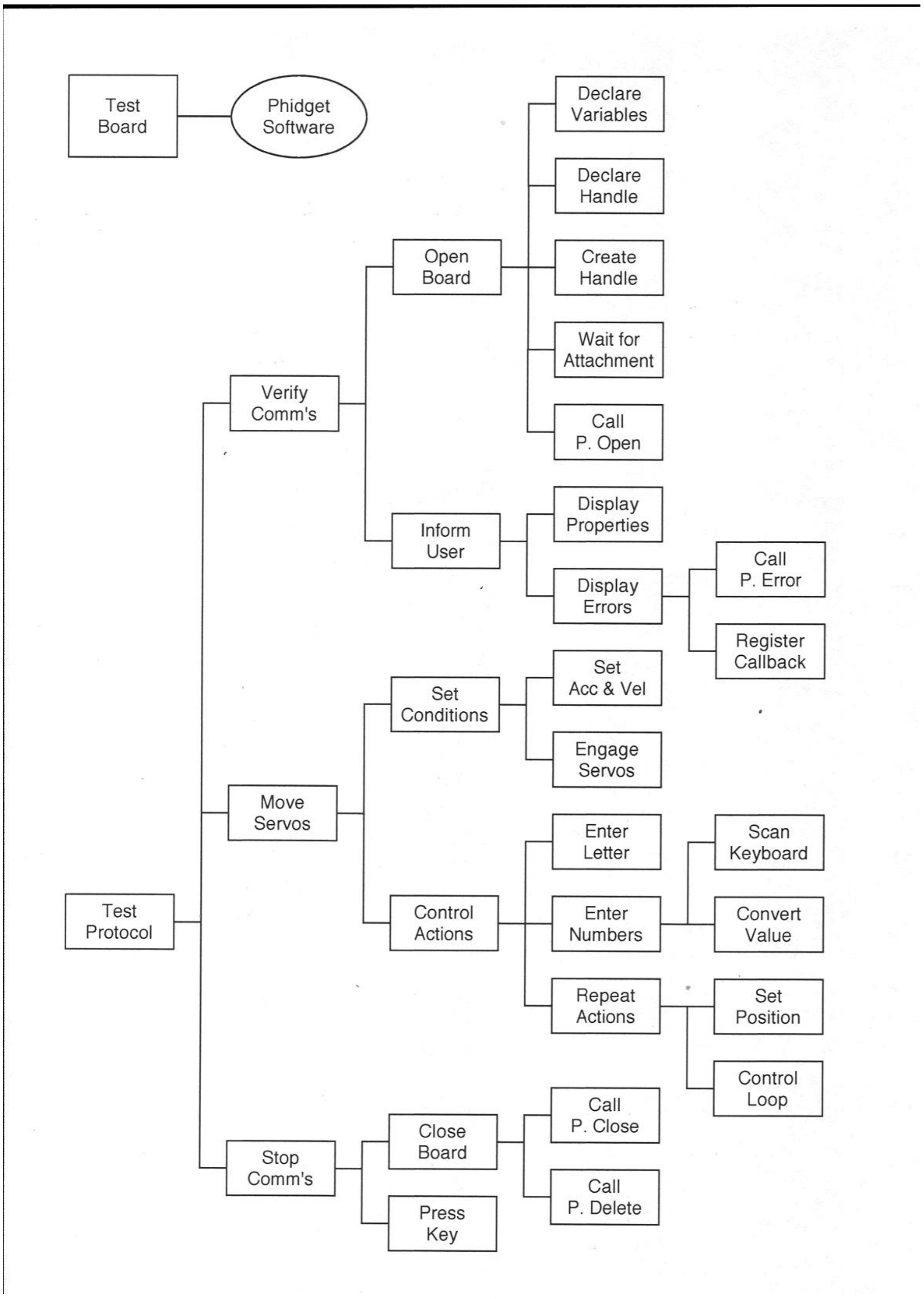

Figure 4. Function Diagram for the Servo Software

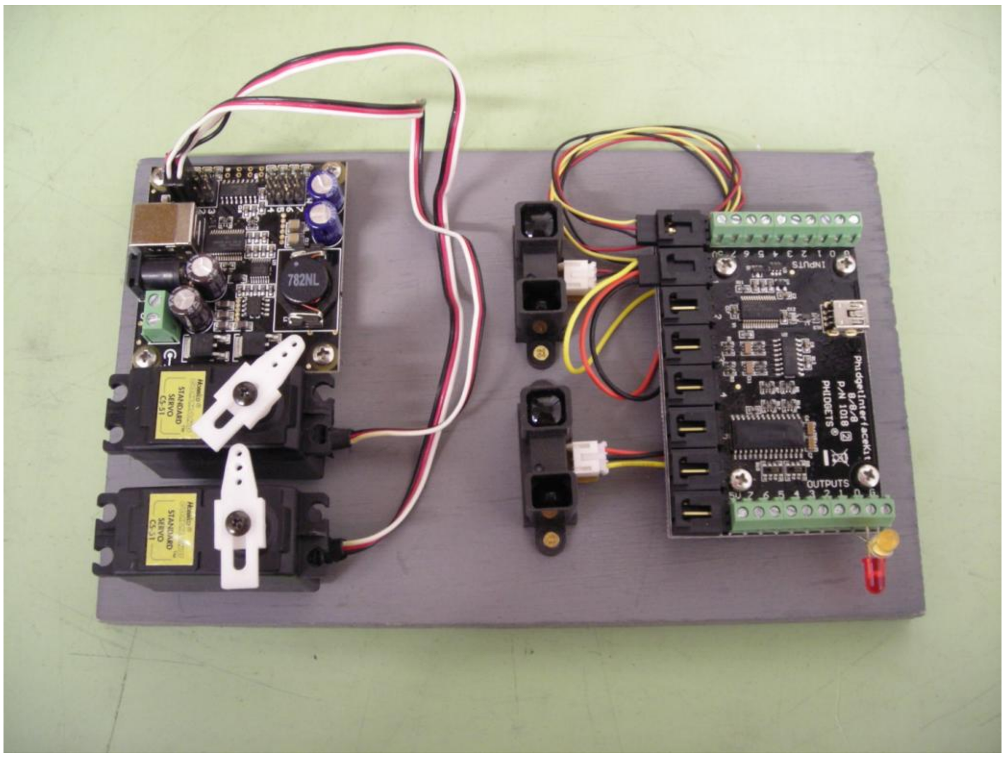

Figure 5. Instructional Board for the Interface Kit and the Servo

References

[1] Bergsman, P., 1994, *Controlling the World with Your PC*, Newness Press, ISBN-10: 1-878707-15-9

[2] Negroponte, N., 1995, *Being Digital*, Vintage Publishing, ISBN-10: 0-679762-90-6

[3] Gleik, J., 2000, *Faster*, Vintage Publishing, ISBN-10: 0-679775-48-X

[4] Podgórski, A., Nedwidek, R., Pochmara, M., 2003, "Implementation of the USB Interface in the Instrumentation for Sound and Vibration Measurements and Analysis", *IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications,* 8-10 September 2003, Lviv, Ukraine, pp 159-163.

[5] Horton, I., 2008, *Beginning Visual C++ 2008,* Wiley Publishing (Wrox imprint), ISBN: 978-0-470-22590-5, pp 358-366.

[6] www.ftdichip.com, 3 November 2011

[7] www.usb.org/home, *Universal Serial Bus Specification*, Revision 1.1, Jan 1998

[8] www.ni.com/labview/, 3 November 2011

[9] www.rockwellautomation.com/rockwellsoftware/design/rslogix5/, 3 November 2011

[10] Bishop, R., 2001, *Learning with LabVIEW 6i*, Prentice Hall, ISBN-10: 0130325597

[11] Zheng, L., 2010, *Ontology-based Knowledge Representation and Decision Support for Managing Product Obsolescence*, Ph.D. Dissertation, Virginia Polytechnic Institute and State University, May 13

[12] Fowler T. C., 1990, *Value Analysis in Design*, Van Nostrand Reinhold, ISBN-10: 0-442-23710-3

[13] Sturges, R.H., O'Shaughnessy, K., & Reed R. G., 1993. "A Systematic Approach to Conceptual Design Based on Function Logic" *Int'l Journ. of Concurrent Engineering: Research & Applications (CERA),* Vol. 1, No. 2, June 1993, pp 93-106.

[14] Ruggles, W.F., 1971, "FAST - A Management Planning Tool", *SAVE Encyclopedia of Value,* Vol. 6, p. 301.