A DATA ACQUISITION AND INFORMATION HANDLING SYSTEM IN ADA FOR ELECTRON SPECTROSCOPY

by

Mats Carlsson and Lars Asplund,
Institute of Physics, Box 530, S-751 21 Uppsala, Sweden.

ABSTRACT

A distributed, real time, data acquisition computer system for electron spectroscopy, ESCA, is presented. The design and implementation in Ada involves windowing, menus, forms, graphical presentation, multitasking and instrumental communication. Our experience using Ada is discussed. Ada has been used in all phases. Data types and packages are presented. It is found that the language is very suitable for scientific purposes.

1. INTRODUCTION

The need for replacing the old data acquisition computer system in recent years has grown. Discussions lead to a series of demands:

- The software must be reusable and portable due to the rapid development in computer hardware. The device dependent parts in the system must be held at a minimum.
- The interaction with the user must be modern and user friendly. Techniques like windowing, function keys, menus and help structures must be included.
- The communication with the spectrometer electronics must utilize a standard.
- The system may be of a single user type with the possibility of several concurrent workstations.
- The data acquisition and user activities in the system must run in parallel.
- The data flow must run freely, but with strong typing, within the system. Possibilities should be given to export data to external applications.

The purpose with this project was to gain knowledge in the following areas:

- How to design and implement a modern, distributed, data acquisition system.
- How the language Ada supports the different phases in system design and project management.
- The behavior of Ada as a real time programming language in a medium sized system.
- The portability of a modular system written in Ada ranging from high level structures, data abstraction and information hiding down to low level device dependent constructs.

2. BACKGROUND

2.1. Ada as a programming language for scientific computing.

Although the main area of interest when designing Ada was embedded systems, the language contains powerful features useful in scientific computing. Features in the Ada language that are important for scientific calculations are the strong typing, overloading, generics, recursion and packages. The specification of Ada permits calls to subprograms, procedures and functions written in other programming languages. It is therefore possible to use the existing investments in scientific software. The separation of the specification and the body in Ada gives the possibility to hide the non Ada implementation of a utility and to give it a new specification. Changeover can be postponed with the possibility of later conversions into portable Ada.

2.2. Ada as a real time programming language.

In the programming of embedded systems the parallelism of the controlled or monitored system must be considered. Software engineering principles indicate that the implementation language should, wherever possible, mimic the structure of the application domain. If the application contains inherent parallelism then the design and construction of the software will be less error-prone, easier to prove correct and easier to adapt if concurrency is available in the design and in the implementation language. Within an Ada program

there may be a number of tasks each of which has its own thread of control. It is thus possible in Ada to match the parallel nature of an application area. Besides the possibility of several concurrent processes there is a need for synchronization and communication.

During execution of a program, events or conditions may occur, which might be considered exceptional. Such conditions are often handled by the operating system with a run time error followed by program termination. This is not acceptable in an embedded system. The software must stand both hardware and algorithmic faults. Ada gives the user the possibility to raise, propagate or handle exceptions within a program unit.

2.3. Computer systems in electron spectroscopy.

Reports about the development of the instrumentation in electron spectroscopy has mainly been focused on components like the electron analyzer, electron lenses, detectors, sample handling and other instrumental hardware. The use of computer software and hardware for the control of the spectrometer has not to a large extent been separately published. It is therefore hard to discuss the general development of the software specific for ESCA spectrometers. A short review of the different techniques that have been used in Uppsala will be given here.

The first computer for spectrometer control in Uppsala was a PDP-15 with the RSX-15 operating system. The memory was only 24 kwords and the programs for recording and plotting spectra were written in assembler [1]. The computer served 4 spectrometers simultaneously. The next generation of computers were the Nuclear Data 6600 systems equipped with two LSI11 processors each. These computer systems have powerful display systems, where spectra very easily can be graphically manipulated. The software, however, is for these machines still written in assembler [2]. Another approach has also been tried where a microcomputer, SWTPC, is used with an interesting hardware device connected between a video camera in the detector and the computer itself [3].

Two other small systems have been used where the software have been written in FORTRAN IV. One of the these systems is a Nuclear Data 6600, which has been connected to a second micro computer system based on a Motorola M6800. The M6800 system does most of the data acquisition while the FORTRAN program in the ND6600 takes care of the disk routines and the terminal
I/O. The communication between the ND6600 and the M6800 is over a serial link [4].

The other system is based on a MINC 11/23, a DEC product, equipped with a LSI 11/23 cpu and RT11 operating system. The detector is in this case connected to the MINC via an IEEE-488 bus. The detector interface has a dedicated microprocessor [5] controlling the hardware timing signals, data buffering and bus communication.

The hardware running the system presented in this report consists of a μVAX II running MicroVMS 4.4 with 6 MB of memory and one 70 MB disk. The detector, power supplies and voltmeter are connected to the system with an IEEE-488 instrument bus. The computer system is also used for calculations and spectrum analysis using CRUNCH [6].

3. SYSTEM DESCRIPTION

3.1. Principles of ESCA.

The instrumentation to be controlled by the new data acquisition system are the ESCA spectrometers developed at the Institute of Physics, Uppsala University. ESCA, Electron Spectroscopy for Chemical Analysis, is an experimantal technique that has reached an increased acceptance by chemists and physicists both at universities and in industry since its inception in the early 1950's [7,8,9]. Under the past years two new spectrometers have been designed, and built, one for the study of gases and solids, and the other for studies of solids and in particular surfaces under ultra high vacuum conditions [10].

Electron spectroscopy reveals fundamental properties of matter. The method is especially usable in characterization of the first atomic layers of a surface. When radiating a sample, electrons are emitted by the photoelectric effect. Different radiation energies can be used to excite the atoms and molecules in the sample. With a UV-radiating source the valence electrons can be investigated, and with a soft X-ray source the valence and the core electrons can be studied.

The speed, or energy, of the expelled electrons depend only on the amount of energy to free the electron from the sample, called the binding energy. Since a sample contains electrons with different binding energies, an electron spectrum will contain a number of peeks or electron lines. The set of characteristic peak positions and relative intensities can be used to determine the elements of presence, and there relative concentrations.

3.2. The ESCA instrumentation.

An ESCA instrument comprises the following main components :

- The sample preparation and introduction system.
- The monochromatic radiation sources, UV and X-ray.
- The preretarding electrostatic electron-lens system.
- The double-focusing hemispherical capacitor analyser.
- The electron-multidetector system.
- The computer hardware and software system.
- The magnetic shielding system.
- The vacuum system.

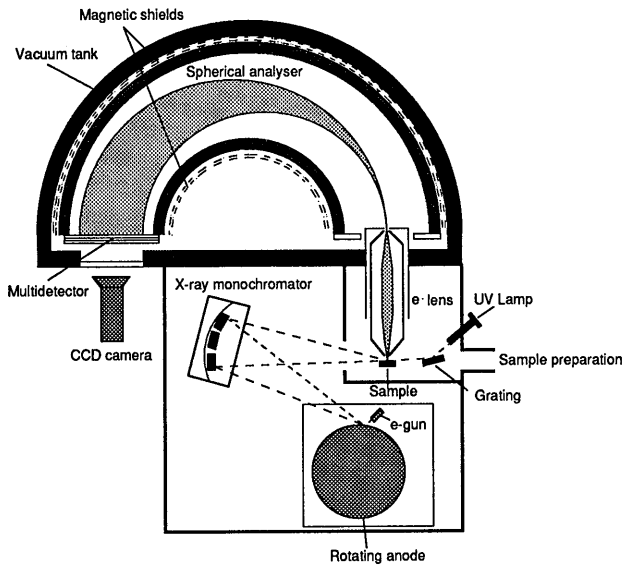

Figure 1. *Schematic drawing of an Esca spectrometer.*

Figure 1 shows the main components in a schematic drawing of an Esca spectrometer.

3.3. Detector.

The electrons expelled, as described above, from the sample are energy resolved at the focal plane of the double-focusing electrostatic analyser. Electron spectrometers with flat focal planes may be equipped with multidetectors based on microchannel plates, MCP's [14]. The microchannel plates amplifies the single electrons into electron pulses.

In the detector system the electron pulses from the microchannel plates are converted into a light pulse by a phosphorous plate. A video camera scans the plate. The resulting peeks in the signal are separated from noise by a discriminator circuit and can be used to increment a counter, counting the dots on each line. For each line the result is added, by a dedicated processor, into a buffer representing the different physical channels in the spectrum. The buffer may later be transferred over the communication link to the computer system.

3.4. Program description.

The new Esca Data Acquisition and Information Handling System, EDIS, is a single user, single instrument data acquisition system. Several concurrent workstations such as text terminals, graphic terminals and plotters can be used. The interaction with the user involves techniques such as menus, function keys, windowing and help structures.

The system can do data acquisition from a spectrometer. The collected data can be stored, viewed, documented, and prepared for further processing. The data to handle is therefore not only spectra but also items closely related to these spectra.

The primary task is the data acquisition and spectrometer control/diagnostic. That part is divided into one for acquisition and one for monitoring. The two subsystems are running simultaneously with other applications in the system allowing the user to evaluate spectra while the spectrometer is running.

The set of parameters, defining a spectrometer run, may be edited in a specialized editor. Old run parameters stored in the data base can be viewed, edited and stored as new run parameters. One set of parameters are called the current run parameters and are used as default when an acquisition is started.

In a documentation editor the user can add information to the spectral data returned from the acquisition system.

Another option for the user is to calibrate the spectrometer and the detector.

A help structure allows the user to retrieve application specific or global help at all levels and applications in the system. The help includes information about the defined function keys.

4. DESIGN AND IMPLEMENTATION

The Esca Acquisition System, EDIS, consists of several parts. Some parts are packages other parts are groups of packages. In the following chapters the design and implementation of the different parts in EDIS are discussed. Some examples of the data structures and program structures are given.

4.1. Data base system.

The data base system of EDIS is a central part for storage and retrieval of information. Almost all of the major data types in the Esca types packages are stored in the data base.

Input and output, IO, is provided in Ada by a set of predefined packages. We soon found difficulties using the IO packages defined in the Ada Reference Manual [18]. The packages for direct or sequential file storage are generic and must be instantiated for all types used. A data base system must handle a number of files, one for each type. Items of record type could easily change in storage size. One example is given by:

```
type CHANNEL is range 0..2**12;
type COUNTS is range 0..1E9;
type E_VOLT is
   delta 0.001 range -10_000.0 .. 10_000.0;

type SPECTRUM_ARRAY is
   array ( CHANNEL range <> ) of COUNTS;

type LOGIC_SPECTRUM (LENGTH:CHANNEL:= 0) is
   record
      STEP: E_VOLT;
      SPECTRUM: SPECTRUM_ARRAY (1..LENGTH);
 end record;
```

One can see that the element LOGIC_SPECTRUM can change in size depending on the field LENGTH. The component of the type COUNTS may be stored in a 32 bit word, or 4 bytes. As the type LOGIC_SPECTRUM mainly consists of the spectrum array, with a size between 0 and 4096, the storage requirement may reach 16KB.

Early tests showed that an instantiation of the generic package SEQUENTIAL_IO reserved the maximum storage size for each element on the file, independently of the actual storage size required to save a particular element. This storage method is powerful as elements on a file easily can be replaced by new larger or smaller ones without problems. In our application items created and stored in the data base, are rarely altered. As the variation in size between different items is large, a method had to be developed where instantiations of different data types could be stored in sequential files, optimizing the file storage size. Items to be stored in the data base are in memory stored as a sequence of storage units, bytes. Knowing the start address and the length of the storage segment, data can easily be transferred to a file handling sequences of bytes. With the attribute 'ADDRESS returning the start address of a variable and 'SIZE returning the number of bits that are occupied by the variable, a generic routine can be used to store and retrieve sequences of bytes. Problems may arise when porting the program to other computers systems that do not represent items as sequences of bytes, or if the run time system is radically changed. Another approach would be to make use of representation specifications for the different types. For simplicity in the implementation we have choosen the former approach.

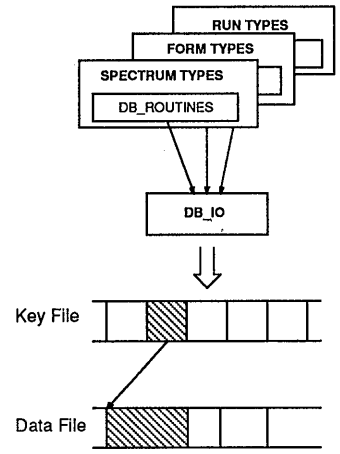

Figure 2. Design of the data base system.

All the types handled by the data base system are referred to by pointers, called access types in Ada. The pointers contains addresses to the data objects. One of the main ideas was to convert these access types to a new type, called segment, declared as a sequence of bytes with variable length, and letting the data base system handle the file storage of this type. A pointer to segments are declared and a generic function for unchecked conversions between the segment pointer and various pointer types are defined. Thus the data base system may handle any item type.

In EDIS we designed one generic package to handle the conversion between the instantiated type and sequences of bytes, and one package to handle the file that stores the items. The separation of the generic converter and the non-generic file handler is necessary as the file handling must be non concurrent and a separation reduces the number of tasks involved.

Figure 2 above shows the schematic design of the data base system. The data base system contains two major packages, DB_ROUTINES and DB_IO.

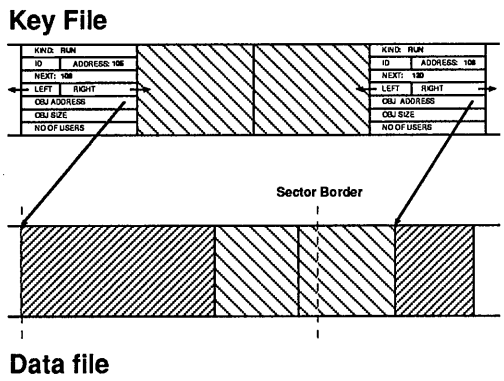

Figure 3. The Key and Data files in the Data Base System.

DB_ROUTINES is a generic package for handling the different data type. The package contains routines for storing, retrieving, locking and deleting single items. Most of the different Esca types packages contain instantiations of this package. DB_ROUTINES make the conversion between the item type and the segment and calls the DB_IO package to complete the operation on the segment.

The data base system uses two different direct IO files to store the information. One is the key file and the other is the data file. The key file is a directory to the data file, also containing references to all different data types objects handled. The key file contains additional information linking the objects in logical groups.

As shown in Figure 3 the elements in the key file contain information about the data type, reference to the data file containing the actual data element and indices to other elements in the key file. A list of deleted elements is held in a key list. This free list may either be used to reuse holes in the data file or used to pack the files to reduce the unused file space.

The data file is divided into sectors, each of equal size. A number of segments may be stored in one sector and one segment may cross the border between two sectors. The reference to the data file contains both the sector index and where in the sector the item begins. When a object is about to be stored the data base system checks the length of the object and allocates the proper size within one or two sectors to store the object efficiently. A copy of the key file is held in memory, in parallel with the key file, giving a 'ram directory'. This parallel structure gives fast access to indices in the data file, thus speeding up the data base transactions. Different structures of the 'ram directory' may be used. When a new item is stored in the data base the key is added to the key file and inserted into the ram directory. The structure of the data base system may seem to be build on a vary low level basis using the representation of the items and thus introduce portability problems. We have deliberately restricted the use of VAX ADA specific packages to enhance the portability. A full implementation of the data base system is running without problems both on a IMB PC AT using MSDOS and a µVAX using VAX/VMS.

As previously described all types handled in the data base system must be access types. The data base systems maps the pointers by assigning every object one unique index. The indices are calculated at the time of creation. Every object must thus have one index with its data base 'name'. Other objects referring to another must have an index, for data base reference, and one pointer, for run time reference. The data base system includes function for creating, saving, rewriting, restoring, deleting elements only by referring to its unique index.

All data base routines have a reference to which data base is to be used. This makes it possible to use several data bases in a system. In EDIS two different data bases are used, one for storing system dependent objects, such as forms, and one user data base for user specific information. The Adabase application may temporary use more then two data bases during data transfer.

4.2. Esca Types.

The EDIS is designed to have free information flow between the different applications within the system. The data base system, described above, is a central unit for storing and retrieving information created and manipulated during different stages in the system use. A number of packages declaring the central data types are specified. These packages are :

- SPECTRUM_TYPES

Containing types handling spectra and items related to these.

- RUN_TYPES

Types containing run parameters and links to spectra.

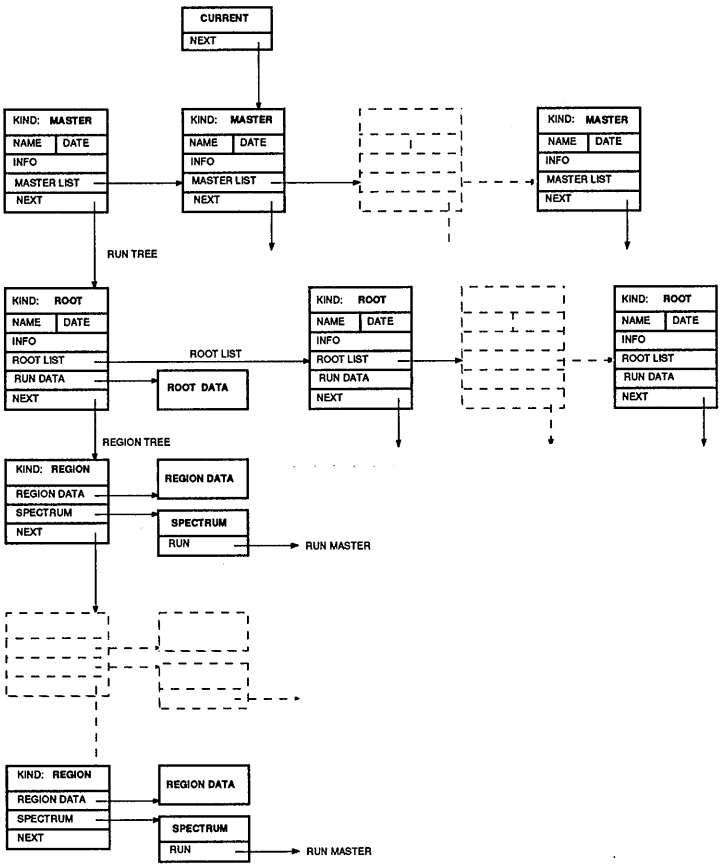

Figure 4. The run tree.

• DETECTOR_TYPES
Types containing parameters for the detector, tvlin table.

• DOCUMENT_TYPES
Containing links to run results not yet documented.

• FORM_TYPES
Containing types handled in the form editor.

The data base is instantiated for a particular type by instantiating the generic DB_ROUTINES package described in the previous part. An application can use the types and within the same package have access to the data base. The Esca types packages adapt its data base handling routines to suit the types involved. A call to the RESTORE routine in FORM_TYPES returns for example not just one form element but a tree representing a complete form.

The strong typing and constraint checking of Ada is very usable in abstracting the different data types used in Esca. The following examples will illustrate the structure of run types and spectrum types. Figure 4 show the structure schematically.

When designing the Esca types large efforts were made to map the structures in the application [11]. Pointer structures were used to represent the dynamic dependencies between the different data types. The structure should also be used by the data base system for searching, as the structures are directly used in the data base. A run in EDIS is described by a set of run parameters. Any number of run parameter sets may be created, stored and used.

A run parameter set is represented as a tree, starting with a master link. The master link contains the run parameter name, the date of birth, generic information, links to the next master and finally links to the run parameter tree. One of the run parameters is regarded as current and is used as default

when the acquisition is started. The different run parameters created in the system may be viewed by traversing the master list and displaying the names.

All run parameters and run results created with a particular parameter set follow the master link in a run tree.

Run results created by the acquisition-subsystem are logically connected to the set of run parameters used to create the result. The run parameters and run results are described in a run list. The first root element is the header for the run parameter list and the other root elements are headers for the different run results derived from the run parameters. All the root elements contain run names and birth dates and generic information. By traversing the run list associated to a single master all run results created from a single set of run parameters may be viewed.

A run consists of one or more regions. Each region is a continuous part in the spectrum and can be scanned a number of times when recorded. In region lists each regions is represented by one element. The region elements contain references to a resulting spectrum.

Both the root element and all the regions refer to data elements. The root data element contains data valid for the whole run. Region specific parameters may be added to region data elements to alter the desired parameters. The result from a run, a virgin spectrum, is associated to each region. The virgin spectrum contains, beside all the spectrum data, a reference to the master used in the creation of the spectrum. The complete run and spectrum structure is viewed in Figure 4.

Further packages are primary type declaring, but do not involve the data base system:

• GKS_TYPES
One part of the Ada GKS system containing all types used to specify the GKS interface.

• HELP_TYPES
Types building the help structure.

4.3. Form Editor.

In EDIS the different applications often needs information entered by the user. For every input situation the user fills in a specific form. The application may then later interpret the result and continue. The design of the editor must be independent of the different forms used, which may be described in some data structure used for forms. This data structure can be stored in the data base system and restored when needed.

The form editor is designed to allow interaction with the using application. When the user fills in a line marked for

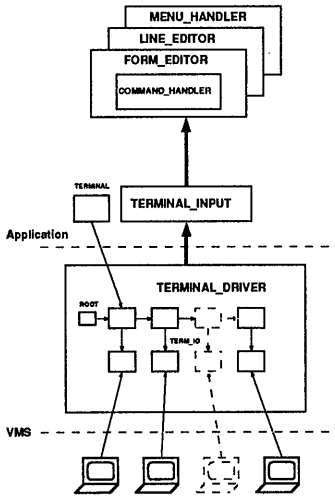

Figure 5. The command input packages.

interaction, the form editor returns to the application. The application may scan the form and perform calculations between different fields.

4.4. Screen.

Techniques like windowing are used in EDIS. The SCREEN package was designed to be used as a utility, handling the windows. SCREEN is a true parallel window handling utility partly implemented in Ada, thus not completely portable to other machines. The interface between the Ada and the non Ada code is well defined and the non Ada code may later be converted into Ada. The non Ada routines used is the run time library routines SMG$,Screen Management Guidelines, in the VAX/VMS operating system.An implementation of the screen package is written in Ada and is running on an IBM PC AT system. This screen package is machine dependent and thus not portable to the µVAX system as it uses the memory mapped screen for fast screen access. The statement 'true parallel' means that not only the 'current' window, but all windows may be addressed and used at any time.

4.5. Command input handling.

Using function keys in the interaction with the user improves the user friendliness of the system. Commands can be given with a single key instead of a command string. When designing the command input handling for EDIS care had to be taken because the use of different terminal types or that the system may run on a personal computer. Function keys on terminals often send a sequence of characters coding the key. One or more lead-in characters signal the presence of a function key stroke, compared to a series of 'normal' key

strokes. The coding of function keys on different terminals often differs. The command input handling in EDIS accepts, at the present state, Digital VT100 and VT200 series terminal and IBM personal computers. The command input handling in EDIS consists of three parts, each with its specific tasks and well defined interfaces to the upper and lower levels. One problem in writing portable code is that the IO operations often are operating system, OS, dependent. The first and OS dependent part, in EDIS, is the TERMINAL_DRIVER package. The TERMINAL_DRIVER handles input, and output, from all the concurrent workstations by assigning a handler task to each workstation. The task type is named TERM_IO. In the TERMINAL_DRIVER package a structure links the terminal driver tasks. Figure 5 shows the terminal driver task data structure.

The TERMINAL_DRIVER is a generic package instantiated for the type of input characters used. In EDIS the package is instantiated for 7-bit ASCII, however it may later be installed for 8-bit characters when needed.

The received characters are buffered and later collected and interpreted by the TERMINAL_INPUT package. This package checks if the strings match any of the predefined character sequence used. If a mach is found, that function key is buffered and later returned to a caller. The character sequences representing the different function keys are sorted so a mismatch can be detected without scanning the whole list. When designing the command input packages we discussed the need for dynamic installation of new terminal types. We found that there was no need for this facility. As mentioned above DEC VT100, VT200 and IBM PC are valid terminal types. However, if a new terminal type is to be installed in the system only the TERMINAL_INPUT package have to be rewritten. In the TERMINAL_INPUT the following function keys are presently defined:

```
type KEY_SORT is (
   ESCAPE,     ILLEGAL,    FIND,
   F6,         F7,         F8,
   INSERT,     F9          F10,
   F11,        F12,        F13,
   F14,        HELP,       KDO,
   Key0,       Key1,       Key2,
   Key3,       Key4,       Key5,
   Key6,       Key7,       Key8,
   Key9,       MINUS,      COMMA,
   PERIOD,     ENTER,      PF1,
   PF2,        PF3,        PF4 ,
   Del,        CR,         IBM_Home,
   IBM_End,    IBM_PgUp,   IBM_PgDn,
   IBM_F1,     IBM_F2,     IBM_F3,
   IBM_F4,     IBM_F5,     IBM_F6,
   IBM_F7,     IBM_F8,     IBM_F9,
   IBM_F10,    IBM_S_F1,   IBM_S_F2,
   IBM_S_F3    IBM_S_F4    IBM_S_F5
   IBM_S_F6,   IBM_S_F7,   IBM_S_F8,
   IBM_S_F9,   IBM_S_F10,  IBM_C_F1,
   IBM_C_F2,   IBM_C_F3,   IBM_C_F4,
   IBM_C_F5,   IBM_C_F6,   IBM_C_F7,
   IBM_C_F8,   IBM_C_F9,   IBM_C_F10,
   IBM_A_F1,   IBM_A_F2,   IBM_A_F3,
   IBM_A_F4,   IBM_A_F5,   IBM_A_F6,
   IBM_A_F7,   IBM_A_F8,   IBM_A_F9,
   IBM_A_F10,  Ctrl_PrtSc);
```

In the design of EDIS we wanted to restrict the use of function key names in the program code, thus facilitate the implementation of future terminal types, and to improve the readability. A generic package, COMMAND_HANDLER, was designed to meet these needs. The command handler package is instantiated with an enumeration type representing the different commands. Different application may instantiate its own command handler to map the specific commands. The function keys and the commands are linked via a command table. All the function keys specified in TERMINAL_INPUT are given a command. The function keys not used are given a No OPeration, NOP, command. The COMMAND_HANDLER was designed to handle keyboard and position sensitive screen input. Only keyboard input is implemented in the current version of EDIS. The following example shows how the line editor specifies the function key to be used:

```
-- Declare the different commands to be used.
type LINE_EDITOR_COMMAND is (
      MOVE_RIGHT,              RUBOUT,
      DELETE_TO_END_OF_LINE,MOVE_LEFT,
      START_OF_LINE,           END_OF_LINE,
      READY,
      INSERT_OVERWRITE_TOGGLE,
      HELP,                    NOP );

package LINE_EDITOR_COMMAND_HANDLER
   is new COMMAND_HANDLER(
      -- The used commands
      SOFT_FUNCTION => LINE_EDITOR,
      -- The No OPeration key
      NO_COMMAND => NOP,
      HELP_FILE => EMPTY_STRING);
use LINE_EDITOR_COMMAND_HANDLER;
-- Link the function keys to the
-- line editor command
TABLE : COMMAND_TABLE :=
      COMMAND_TABLE'(
        TERMINAL_INPUT.UP => START_OF_LINE,
        TERMINAL_INPUT.DOWN => END_OF_LINE,
        TERMINAL_INPUT.RIGHT => MOVE_RIGHT,
        TERMINAL_INPUT.LEFT => MOVE_LEFT,
        TERMINAL_INPUT.ENTER => READY,
        TERMINAL_INPUT.PF1 =>
             INSERT_OVERWRITE_TOGGLE,
        TERMINAL_INPUT.HELP |
        TERMINAL_INPUT.PF2 => HELP,
        TERMINAL_INPUT.REMOVE =>
             DELETE_TO_END_OF_LINE,
        TERMINAL_INPUT.PF3 =>
             DELETE_TO_END_OF_LINE,
        TERMINAL_INPUT.DEL => RUBOUT,
        TERMINAL_INPUT.CR => READY,
        TERMINAL_INPUT.IBM_F10 => READY
        others => NOP );
```

The command handler also controls the help structure in EDIS. The 'HELP' key may be defined to return a command. Then the application program must handle the help by its own. If the 'HELP' key is not defined then the command handler sends a help identifier to the help system to identify the position in the program where help was required. The help identifier is one of the parameters when instantiating the command handler, called the HELP_FILE. The functionality of the COMMAND_HANDLER is event controlled. All calling routines may investigate the type of input, command, character or control character, before reading or rejecting the actual input data.

Characters, but not strings, may be retrieved form the command handler. Further utility levels serving the applications with line editors for string, enumeration and numeric input can thus be build using the command handler.

4.6. Menu Handler.

In the communication with the user menus are used. A menu is a collection of items in which the user may select one. To prevent the spreading of detailed menu handling all over the system, a generic menu handler was designed. The menu handler is instantiated with an enumeration type representing the range and names of the different menu alternatives to be handled and a NOP_KEY function. An instantiation called the STANDARD_MENU_HANDLER instantiated with NATURAL is often used in EDIS. To use this utility the application builds an array containing the menu texts. When called the menu handler fits the different alternatives into the supplied window, with the constraint to minimize the columns. If some of the alternatives do not fit within the window then the window can be scrolled to let the user view the remaining menu alternatives. A default choice and two headers are also supplied in the call. The following example shows the use of the menu handler.

```
declare
   -- Declare a menu containing
   -- the different choices.
   MY_MENU : MENU_TYPE ( 1 .. 3 ) := (
                        TO_TEXT( "Start" ),
                        TO_TEXT( "Stop" ),
                        TO_TEXT( "Quit" ));
   -- Users choice.
   CHOICE : NATURAL;
begin
   -- Present the menu and let the user select
   -- among the alternatives.
   CHOICE := MENU_SELECT(
      MENU => MY_MENU,
      HEADER => TO_TEXT( "Select one item");
      SUPER_HEADER => TO_TEXT( "EDIS");
      DEFAULT_CHOICE => 1,
      WINDOW => MY_WINDOW );
   -- Handle the selected alternative here.
end;
```

4.7. Graphic utility and GKS.

Spectral information in EDIS is presented in graphic form to the user. A standard graphic utility called GKS, Graphical Kernel System, is used.

The graphic utility of EDIS and the window handling had to be connected in some way. We decided to choose a solution where the graphic utility is a part of the window handling. Graphics can be presented within a text window. For implementation simplicity we restricted the graphic window to never be covered or occluded by another window. The package GRAPHIC_SCREEN handles these tasks. This package communicates with SCREEN and GKS defining windows and viewports matching the screen windows. Some problems arises since the GKS and SCREEN packages use different modes of the terminals, thus text and graphic output may never run in parallel. Synchronization with the screen package is also one of the tasks for the GRAPHIC_SCREEN package.

The package SPECTRUM_GRAPHICS is designed for presenting spectrum graphics. Information about the start and end energies and the maximum number of counts are presented in a text window and spectra in a graphic window.

4.8. Spectrometer packages and IEEE-488.

The hardware for the spectrometer is based on a number of distributed processors all connected to the IEEE-488 bus. The distributed system controls a video camera used as detector, power supplies and a voltmeter. In EDIS all the distributed services are mapped with packages, the spectrometer packages, having the same functionality as the distributed hardware shown in Figure 6.

The package VIDEO_INTERFACE gives commands to the video camera interface and reads data from the interface. Data are returned to the program in the form of logic spectra and all transformations are performed within the package. A package called VOLTAGE_SOURCE controls the power supplies used. This package may serve any number of power supplies. All the spectrometer packages uses the IEEE bus via the IEEE_488 package.

4.9. Acquisition subsystem.

The central part in the spectrometer system is the acquisition subsystem, which controls the spectrometer, collects the spectral data and stores the data in the data base. In the specification of EDIS one of the demands was that data acquisition, acquisition monitoring and user activities in the program must all run in parallel. In the design of real time systems it is essential to detect the behavior of the different parts in the system. The use of Ada tasks is one tool in real

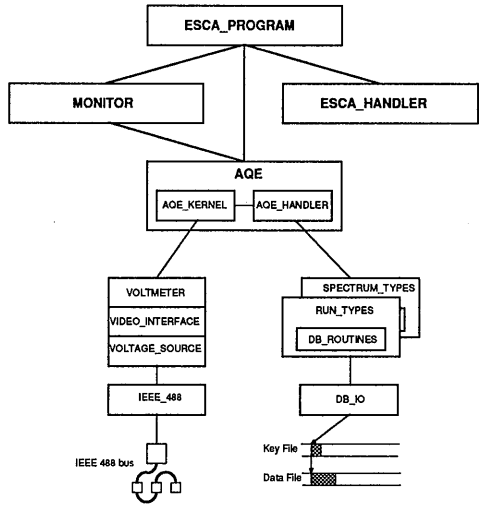

Figure 6. The ESCA_PROGRAM.

time programming. In the acquisition subsystem, AQE, the following design problems arise.

To minimize the data acquisition run time the program must serve the dedicated detector hardware constantly, starting the acquisition and later receive the spectral data. At the same time the collected data must frequently be saved in the data base, assuring that data is not lost due to a general failure.

The acquisition monitor must make asynchronous calls to the AQE to retrieve the present AQE status. The AQE was designed to include both the user interface, the acquisition monitor interface and the data acquisition kernel, handling the spectrometer and the data base storage. Figure 6 shows the dependencies in the AQE.

Two tasks are contained in the AQE: AQE_HANDLER and AQE_KERNEL. The AQE package specification contains routines for both user communication and acquisition monitor calls. At the start of a run the AQE restores the run parameters to be used and passes the parameters to the AQE_-HANDLER task. The AQE_HANDLER serves the acquisition monitor, the AQE_KERNEL and handles the calls to the data base.

When the AQE_HANDLER receives a set of run parameters, it interprets the parameters and starts the AQE_KERNEL, passing region information to the task. Collected data are frequently retrieved form the AQE_-KERNEL and saved in the data base. The AQE_KERNEL handles the spectrometer and commands the dedicated hardware connected to the spectrometer. As previously described the distributed system communicates over the IEEE_488 bus and the system is mapped with specific packages within EDIS.

The AQE_KERNEL task is started by passing information about a region run. The AQE_KERNEL performs the acquisition by stepping the region. For each step the AQE_-KERNEL sets the voltage, starts the data acquisition and receives data from the detector. After a step is completed the AQE_KERNEL checks if AQE_HANDLER is waiting to receive the spectral information collected. This check is performed by a timed accept.

When a region is completed the AQE_KERNEL waits for new information, describing the next region to be recorded. At any time, after a completed step, the AQE_HANDLER can be stopped or terminated. At termination, the AQE_-HANDLER returns the collected data. At stop the AQE_-HANDLER stops the run and waits for a resume command. The user may, using this facility, temporary stop the data acquisition.

When writing concurrent real time programs targeted for a single processor system, the questions of process priority arises. How to order the different processes, tasks, in a system, giving them the right priority? What happens if two tasks have the same priority, which one is running and which one is suspended? May an important task be starved and never able to execute? All these questions must be considered. The ARM [12] gives no guidance on how tasks should be interleaved on a single limited processor. In order to minimize tasks switching many implementations will choose the following algorithm. A task, once it is executing on a processor, will continue to execute until it is no longer in the state of 'executable'. Task switching will not occur until the

executing task will change its state by :

- completing its execution,
- executing a delay statement,
- executing an entry call,
- elaborating a sub-task,
- executing an accept statement upon witch no entry call has been made,
- executing a select statement in witch there is no else part and no outstanding entry calls on any open select alternative.

An interrupt entry into another task is the only exception to this rule. When assigning priority to the involved tasks the following rule, stated in the ARM, stands.

"If two tasks with different priority are both eligible for execution and could sensibly be executed on the same physical processor and some physical resources, then it cannot be the case that the task with the lower priority is executing while the task with the higher priority is not". (ARM 9.8.4)

In VAX ADA, task switching is performed in the ways described above. When switched, tasks with the same priority are executed in a first-in-first-out order. To allow additional control over the task scheduling, VAX ADA provides the pragma TIME_SLICE. This pragma causes the task scheduler to limit the amount of continuous time given to a task when other tasks of the same priority are also eligible for execution. This pragma is a non-portable feature.

In EDIS the AQE tasks is given the highest priority, letting the data acquisition run at full speed. The AQE_KERNEL have a slightly higher priority than AQE_HANDLER. This allows the AQE_HANDLER frequently store collected data in the data base, while the AQE_KERNEL is waiting for the detector to transmit data. Next in the priority list is the IO tasks, handling the user terminal IO operations. It is desirable that user input is acknowledged by the applications as fast as possible. Some of the tasks used in user applications, like the acquisition monitor and the clock, run with a slighter higher priority than the main program

4.10. Acquisition Monitor.

The acquisition monitor is a user application showing the state of the data acquisition subsystem. The acquisition monitor is also used to view the collected data in the various regions under investigation. The monitor runs in parallel with other activities in the system.

The acquisition monitor was designed to work in two modes. One mode is the interactive mode where the user is communicating with the acquisition monitor, investigating the state of the data acquisition subsystem and viewing spectral data. The other mode is the background where the monitor frequently updates a window, presenting the current status of the acquisition subsystem. In the background mode the user is free to run any application in the foreground and still view the monitor information.

The acquisition monitor, MONITOR, calls the AQE package for both status and spectral data. While calling the AQE package the caller may hang waiting for the AQE_HANDLER and AQE_KERNEL to respond to a call. Utilities like the SPECTRUM_GRAPHICS are used to present the spectral data to the user. In the monitor a task was designed, handling the concurrency both in the foreground mode and in the background mode. This task is named the AQE_MONITOR task. This task is both started and stopped by commands form the user, given in the MONITOR package. When the AQE_MONITOR is started a window is passed to the task to be used for output.

The different windows used by the user applications in EDIS is controlled by the ESCA_HANDLER package. This package routes the calls form the main procedure, ESCA_PROGRAM, down to the user application packages. In Figure 6 the ESCA_PROGRAM and the ESCA_HANDLER are shown. The package handles three windows, the main window, the clock window, and the monitor window. In the main window all the different applications, except the monitor, are executing. The monitor and clock window holds these specific applications. The Esca handler changes the size of the different windows when different applications are called. The Esca handler also manages the stacking order of the these windows. The clock window is always the top window, followed by the monitor window and finally the main window. The Esca handler is the initiator and terminator in EDIS. When EDIS starts, the Esca handler package opens the data bases, activates the IEEE 488 bus and shows any system messages to the user.

5. CONCLUSIONS

When this project started the use Ada as a language for software production had merely started. The group had very little experience in real time programming in Ada.

Three different Ada compilers have been used (TELESOFT-ADA, Alsys Ada 1.0 - 3.2, and the VAX ADA 1.0 - 1.5). There is a significant increase in productivity using the later versions of the compilers. The TELESOFT-ADA compilers we used in 1985 could hardly be used for learning Ada. The lack of commercially available software components is still a drawback, comparing Ada to other high level languages, when implementing small to medium sized systems. This has changed, to some extent, in the recent year. Designing and implementing nearly all basic utilities in EDIS has slowed down the development. Using Ada as a

language for design and implementation of a modern, distributed, data acquisition system is however a good choice, becoming better and better. The modularity, reusability and the readability are great advantages compared to FORTRAN. In a laboratory environment an acquisition system must be easy to modify. New measurement methods and equipment must be incorporated in the system by scientists not primary involved in the development of the system. The modularity of the system, if designed in a proper way, may support quick modification.

The behavior of Ada for support in system design and project management is primarily found to be a question of compiler environment. Ada programming support environments are now becoming available. These environments include, beside compilers, editors and debuggers, design aids, configuration managers and project coordination tools. This field often called CASE, Computer Aided Software Engineering, is desperately needed and fast growing. Automated software development is one way to fight the rising cost and extended delays called the software crisis [13]. Ada itself with its specification-implementation separation gives the basic support for central design and decentralized implementation. The standardized Ada Programming Support Environment, APSE, may not be presented in a number of years, as the field is not yet convergent.

The VAX ADA compiler with its ACS library handler is a manageable system for medium sized projects.

The portability of Ada programs is a subject for long discussions. We have found that the low level IO handling nearly always ends up in non-portable code, as was the case with former high level languages. The tools for data abstraction and information hiding give ways to restrict the spreading of non-portable features over the whole system.

The non-portable features may then be rewritten to fit the new target system. The standardized tasking mechanism in Ada gives a way to use parallel structures in a portable way. The upper utility and application levels may thus freely be ported.

The VAX ADA compiler provides a set of non-portable features. This is possible and still following the ARM. A command in the ACS system lets the user check the portability of a Ada program complied with the VAX ADA compiler. It is often tempting to use the nice non-portable features in VAX ADA, and this is in some cases done, hopefully with care.

The behavior of Ada as a real time programming language in a medium sized system must be regarded as good. The Ada language gives the possibility to design and follow the nature of the system to be controlled.

6. REFERENCES

[1] T.Bergmark and E. Basilier,
Uppsala University Institute of Physics Report,
UUIP-846, 1974

[2] E. Basilier,
Uppsala University Institute of Physics Report,
UUIP-1022, 1980.

[3] J. Nordgren, R. Nyholm and L. Pettersson,
Ann. Reg. Sci Acta Upsaliensis, 23, 96, 1980.

[4] Hans Veenhuizen,
Uppsala University Institute of Physics Report
UUIP-1100, 1984

[5] L. Asplund,
Uppsala University Institute of Physics Report,
UUIP-1073, 1983.

[6] K. Siegbahn, C. Nordling, A. Fahlman, R. Nordberg,
K. Hamrin,J. Hedman, G. Johansson, T. Bergmark,
S.E. Karlsson, I. Lindgren and B. Lindgren,
ESCA Atomic,molecular,solid state structures studied by means of electron spectroscopy,
Almqvist & Wiksell, 1967.

[7] K. Siegbahn, C. Nordling, G. Johansson, J. Hedman,
P.F. Hedén, K. Hamrin, U. Gelius, T. Bergmark,
L.O. Werme, R. Manne and Y. Bear,
Esca applied to free molecules,
North-Holland Publishing Company, 1969.

[8] B. Lindberg, Kemisk Tidskrift 6, 1983.

[9] U. Gelius, L. Asplund, E. Basilier, S. Hedman,
K. Helenelund and K. Siegbahn
Nucl. Instr. and Meth. 85-117 1984.

[10] E. Basilier,
Uppsala University Institute of Physics Report,
UUIP-1021, 1980.

[11] L. Asplund, K.Helenelund, M. Carlsson
and A. Gisslén,
Uppsala University Instituteof Physics Report
UUIP-1137, 1987.

[12] *Reference Manual for the Ada Programming Language, (ANSI/MIL-STD-1815A),*
Ada Joint Program Office, Department of Defense,
Washington, D.C., 20301, 1983.

[13] W. Suydam,
Computer Design, January 1, 1987.