Acta Cryst. (1980). A**36**, 979–989

XTAL: New Concepts in Program System Design

BY S. R. HALL

Crystallography Centre, University of Western Australia, Nedlands, Australia

AND JAMES M. STEWART AND ROBERT J. MUNN

Department of Chemistry and Computer Science Center, University of Maryland, College Park, Maryland, USA

(*Received* 14 *February* 1980; *accepted* 18 *June* 1980)

Abstract

Program systems are used extensively for crystallographic computing. Most existing systems have problems with language incompatibility, computational inefficiency and poor adaptability. This is because many desirable features demand opposing strategies in system design. Systems development is also complicated by the increasing number of computer types and sizes capable of doing crystallographic calculations. This, coupled with the widening aspirations of crystallographers, is making the task of developing an efficient and comprehensive program system increasingly difficult, particularly with conventional Fortran programming methods. The XTAL system is being developed with a different approach to these problems. The distribution of XTAL source code is in the RATMAC preprocessor language, the use of efficient machine-specific code is encouraged, and a much more flexible directory-type file structure is used for archive data files.

1. Introduction

Distributed program systems are now used extensively by crystallographers for reasons of convenience, economics, and because of a general reduction in the development of in-house software. The continued growth in 'production' type crystal structure analyses also means that many program users have little formal crystallographic training and are dependent on external sources for up-to-date software.

Most existing program systems do, however, have serious deficiencies. Features such as transportability, generality, computational efficiency and flexibility often demand opposing design strategies, despite the use of Fortran as the source language. As a result, systems optimizing computational efficiency often invoke machine-specific code which seriously affects their transportability. Others concentrate on transportability by using only a subset of instructions common to most Fortran compilers and this often degrades relative computational efficiency on machines where powerful machine-specific Fortran features are available. In the same way, programs designed for a wide range of structure types and sizes are often inefficient for specific problems and relatively difficult to adapt or modify. It is not uncommon for a program system to sacrifice one feature for another or to compromise on them all.

The problems facing future systems programmers will not be simpler. Rapid advances in computer technology have spawned an impressive array of machine types and sizes which are capable of doing crystallographic calculations. As a result the degree of transportability and/or efficiency possible with Fortran systems a decade ago is just not possible now with scores of manufacturers and hundreds of machine types. Future program systems must be able to adapt to these hardware changes, and the accompanying operating system software, if an acceptable level of transportability and efficiency is to be achieved.

The single most important obstacle to improving program systems is the Fortran programming language itself. The original arguments for Fortran as a system language, namely transportability, simplicity and adaptability, have been eroded away in recent years. Why not then use the other high-level languages such as Algol, PL1, APL or Pascal? All have more powerful and logical command structures. The answer is that not one of these languages is as widely accepted as Fortran, or have compilers which produce highly optimized code.

The importance and the inadequacy of Fortran has given birth to a new breed of languages, the Fortran *preprocessors*. Preprocessor languages have structured features similar to languages such as Pascal, but differ from these in that they are not translated (*i.e.* compiled) directly into machine code. Rather, the preprocessor language is 'preprocessed' into another language, such as Fortran, and this in turn is compiled in the normal way. In one respect the development of preprocessors

0567-7394/80/060979-11$01.00

appears to be a retrograde step. There is one more stage to the implementation process, and one more language to be implemented. However, these drawbacks are small compared to the gains to be made elsewhere. For this reason the preprocessor language RATMAC (Munn & Stewart, 1978, 1979) is used with the XTAL system. The implications of RATMAC for XTAL are discussed in § 2.

It is neither possible nor economical for all the components of a major program system to be resident in core at one time. An important consideration in the design of a program system therefore is the method of partitioning a calculation into convenient logical units. Many individual system features are dependent on how these partitions are linked, and how they are to be organized in core at execution time.

The three principal linking methods are: 'stand-alone' routines which communicate *via* an independent set of data files; program 'overlays' called from a controlling root element (the 'nucleus') which remains resident in memory; or, in the case of machines with VMS (virtual-memory operating systems), a contiguous block of routines which are paged into core as required for execution. Understanding the properties of these linking procedures is clearly important if a program system is to be accessible to a range of machine types and operating systems. Considerations that have gone into the structuring and partitioning of subroutines in the XTAL system, and the provision of different linking procedures, will be discussed in § 3.

For program systems capable of separate calculations there must be some mechanism for controlling which, and in what order, are performed. At the simplest level are those program systems that execute as stand-alone routines. In this case control is exercised by the user *via* the local command language. For large systems operating in an overlay or a VMS environment, the control function must be managed internally by one of the XTAL system routines, and directed by the user *via* the input data stream. These routines are referred to as the system 'nucleus'. Typically, the nucleus routines are responsible for sifting the input data stream for control parameters and acting upon them accordingly. The nucleus approach has the double advantage of concentrating the essential 'driver' routines in one place to minimize redundancy, and, most importantly, it centralizes routines that often pose implementation difficulties. The use of the nucleus concept in the XTAL system will be discussed in § 4.

An area that frequently causes implementation difficulties is line input–output (I/O). I/O instructions represent the essential interface between the program and the user and must as a consequence provide maximum flexibility. Such flexibility inevitably involves some form of machine specificity and this, as we have stated, is not conducive to transportability. The lack of standards for Fortran I/O instructions such as BUFFERIN/BUFFEROUT, DECODE/ENCODE, binary READ/WRITE, formatted READ/WRITE, PRINT/PUNCH and all the different ways they are parameterized is a strong indictment of the computer industry. The number of parameters and rules associated with the FORMAT statements of different manufacturers is almost limitless. In most systems, therefore, line input–output code is very difficult to make transportable. With the XTAL system specificity is not such a problem because the RATMAC language permits machine-specific instructions. However, line input–output poses difficulties in other respects. Fortran input–output functions are usually handled by a series of run-time library (RTL) routines. These routines, because of their general capabilities, are relatively large and slow. In the XTAL system the RTL routines for Fortran I/O are replaced with small nucleus subroutines which are much more specific to crystallographic needs. Then at the option of the implementor the Fortran RTL may be eliminated. Details of this are given in § 5.

The design of crystallographic programs depends largely on the methods of storing and manipulating data. The large variations in size and nature of crystallographic data make their proper management absolutely crucial. It is generally accepted that the most efficient method of storing data in a program is as a single one-dimensional open-ended array (Stewart, 1976*b*). Data of different types are packed into this array with appropriate markers set to identify the boundaries. Use of these markers facilitates rapid and easy access to the data and permits memory to be allocated only as required. Memory-allocation procedures of either the 'static' type (fixed partitions or once-per-calculation) or 'dynamic' type (during a calculation) are increasingly important with the advent of multi-task variable-partition operating systems. This is because they encourage efficient and economical use of resources. Details of data-management procedures in the XTAL system are given in § 6.

One other aspect of the XTAL system that will be discussed here is the structure of the 'archival data file' used to store crystallographic information between calculations. Because the form and size of crystallographic data can vary greatly from problem to problem and from calculation to calculation, it is difficult to have an efficient, yet flexible, structure for this file. Most data files have a fixed-sequence type of format which works well for an 'average' type of analysis, but is inefficient, and occasionally even inoperable, when problems depart too far from normal. Archiving data associated with either a protein analysis or an accurate electron density study is generally not feasible with the fixed-sequence data files of existing systems. For this reason the XTAL system has adopted a new type of 'directory-driven' file that adapts well to a range of problem types. Details of this file are given in § 7.

2. The RATMAC preprocessor

The RATMAC language (Munn & Stewart, 1978, 1979) is a combination of the structured high-level language *RAT*FOR and the in-line editor *MAC*RO, both of which were developed by Kernighan & Plauger (1976). Advantages in adopting RATMAC as the distribution and implementation language of the XTAL system have already been reported (Stewart & Munn, 1978).

Using RATMAC as the source language, the XTAL system is implemented in a two-stage process. The distributed RATFOR/MACRO instructions are converted into Fortran using the RATMAC preprocessor, and the Fortran is then compiled into machine code. At first sight this two-stage process appears to complicate XTAL implementation rather than simplify it. There is also a need to implement the RATMAC preprocessor itself at each installation. Experience has shown, however, that these factors are minor compared to the overall gains possible through use of the preprocessor. The preprocessor is non-proprietary (*i.e.* no commercial copyright) and is supplied with the XTAL system in two versions. The first is a non-optimized 'bootstrap' program written in ANSI Fortran, that can be compiled directly on most machines. The second version is the RATMAC preprocessor which is written in RATFOR/MACRO ready to be processed using the bootstrap program. This in turn can be compiled to form an optimized working preprocessor program by invoking machine-specific features.

The advantages offered by RATMAC are best illustrated by describing the properties of its component parts, RATFOR and MACRO.

RATFOR (an acronym of RATional FORtran)

This is a high-level language similar to Fortran except for the structured control features that enable 'top-down' programming. Most Fortran instructions are acceptable to the RATMAC preprocessor and are passed to the output file unaltered. However, unlike Fortran, the use of statement numbers and of GO TO's is discouraged. Calculations are controlled with the instructions listed in Table 1.

In general, control instructions act upon a block of statements bounded by the braces { and }. Conditions for the IF, ELSE IF, WHILE and UNTIL instructions are similar to those for the Fortran logical-IF type. An important advantage of RATFOR over Fortran is that it promotes a more logical organization of programs and produces code in which the essential algorithmic structure is apparent. This is best illustrated with an example. Table 2 shows a typical *bubble sort* algorithm written in both RATFOR and Fortran.

The two-loop (one backward, one forward) structure of a bubble sort is largely obscured in the Fortran code.

Table 1. *RATFOR control instructions*

Conditional	

if (⟨condition⟩)
{
⟨statement block⟩
}
	A chain of successive logical conditions. The ⟨statement block⟩ associated with the first *true* ⟨condition⟩ is executed. If no *true* ⟨condition⟩ is found the ⟨statement block⟩ associated with the trailing else is executed.

else if (⟨condition⟩)
{
⟨statement block⟩
}
⋮	

else
{
⟨statement block⟩
}	

Looping	

while (⟨condition⟩)
{
⟨statement block⟩
}
	While the ⟨condition⟩ is *true*, ⟨statement block⟩ is repeatedly executed.

repeat
{
⟨statement block⟩
}
	⟨statement block⟩ is executed 'forever'.

repeat
{
⟨statement block⟩
}
until (⟨condition⟩)
	⟨statement block⟩ is repeated until ⟨condition⟩ becomes *true*. The ⟨condition⟩ test is at the loop end.

for (⟨initialization⟩;⟨condition⟩;⟨reinitialization⟩)
{
⟨statement block⟩
}
	After ⟨initialization⟩ is executed a 'while' loop is set up based on ⟨condition⟩; at the end of each loop the ⟨reinitialization⟩ statement is executed.

do ⟨loop limits⟩
{
⟨statement block⟩
}
	Equivalent of the familiar Fortran DO (without statement labels).

More details of these structures and their uses can be found in Kernighan & Plauger (1976) and Munn & Stewart (1978, 1979).

In contrast the structured RATFOR code is easier to read. RATFOR encourages the programmer to write well organized code that consequently is easier to debug and modify.

MACRO

The structured control features of RATFOR are important to the XTAL system, but it is the MACRO facility of the RATMAC preprocessor that offers the greatest gains towards the system objectives already discussed. In simple terms, a macro provides for in-line substitution and *global editing* of the Fortran code. As the name 'macro' suggests its principal purpose is to replace repeated blocks of code with a single name, much as a *FUNCTION* subprogram does in Fortran. However, unlike a *FUNCTION*, a macro does not have to be executable and is only a recognizable entity *before* RATMAC processing.

Table 2. *Comparison of RATFOR and Fortran for the bubble sort subroutine*

(*A*) RATFOR

```
subroutine rsort (V,n)
real V(n)       # array of real numbers to be sorted
logical notend # notend is set false if a pass
               # produces no interchanges
#
notend=.true.
for (i = n; i > 1 & notend; i = i - 1)
  {
  notend=.false.
  for (j = 1; j < i; j = j + 1)
  if (V(j) > V(J + 1)) # elements out of order
    {
    notend=.true.
    R = V(j)              #
    V(j) = V(j + 1)       # interchange elements
    V(j + 1) = R          #
  } # end of j loop
} # end of i loop
return
end
```

(*B*) Fortran

```
      SUBROUTINE FSORT (V,N)
      REAL V(N)
      LOGICAL NOTEND
      NM1=N-1
C
C---- CHECK FOR N.LE.1
C
      IF (NM1.LE.0) RETURN
C----SETUP REVERSE LOOP N TO 2
C
      DO 2 L=1, NM1
      I=N-L
      NOTEND=.FALSE.
      DO 3 J=1,I
      IF (V(J).LE.V(J+1)) GO TØ 3
      R=V(J)
      V(J)=V(J+1)
      V(J+1)=R
3     CØNTINUE
      IF (.NOT. NOTEND) RETURN
2     CØNTINUE
      RETURN
      END
```

A macro instruction is *defined* as

MACRO: (name:, definition),

where 'name:' is the name by which the macro is referenced in the RATMAC code, and 'definition' is a collection of programming steps. A macro name may have up to nine arguments, *i.e.* presented by the combination $\$n$, where n is the argument number. For example,

MACRO:(BITSWORD:,36),
MACRO:(BITSCHAR:,8)

and

MACRO:(IFIX:,INT(SIGN(0.5,$1)+$1))

define the bits per word, bits per character and real to integer rounding function for a given machine. On preprocessing, every reference to BITSWORD:, BITSCHAR: and IFIX:(X) in the RATMAC code generates a 32, 8 and INT(SIGN(0.5,X) + X) respectively in the Fortran code.

A particularly good example of a substitution macro is the XTAL system common macro, SYSCOM:. SYSCOM: is defined at the beginning of the XTAL code and thereafter programs using the system common block include only the statement SYSCOM:. This type of use results in simpler, more concise and more legible programs, and moreover ensures that any changes in the system common block are *global* changes.

Macros with 'dynamic' properties are also possible. For instance, we would replace the last line in the RATFOR bubble sort subroutine in Table 2 with the statement, SWAPIA:(J,J+1) where this macro was previously defined as

MACRO:(SWAPIA:,R=V($1); V($1)= V($2); V($2) = R).

This macro generates different code according to the values of the arguments $1 and $2. For instance, the statement SWAPIA:(15,IPT+2) in the RATFOR code will appear in the Fortran code as the three lines

R = V(15) V(15) = V(IPT+2) V(IPT+2) = R.

Machine-specific code can also be introduced into a program system using macros. For example, a machine-specific bit-string mover could be incorporated as

MOVEBITS: (⟨word FROM⟩, ⟨bit N⟩, ⟨word TO⟩, ⟨bit M⟩, ⟨bit length NBITS⟩)

with requests that a bit string of length NBITS, starting at bit-position N of word FROM, be moved to word TO, starting at bit-position M. The definition of this macro for different machines is

Univac 1108: MACRO:(MOVEBITS:,FLD(35-$4,$5,$3) = FLD(35-$2,$5,$1))

CDC Cyber73: MACRO:(MOVEBITS:,CALL STRMOV($1,$2,$5,$3,$4))

DEC VAX11/780: MACRO:(MOVEBITS:,CALL LIB$INSV(LIB$EXTZV ($2,$5,$1),$4,$5,$3))

PE 8/32: MACRO:(MOVEBITS:,$3 = IEOR($3,ISHFT(IEOR (ISHFT(ISHFT($1,32-$5-$2), $5,-32),ISHFT(ISHFT($3,32-$5-$4),$5-32)),$4))).

Input–output control is one of the many other areas where macros are useful for machine-specific substitution. The Fortran non-buffered READ instruction varies considerably from machine to machine and

could be conveniently applied in a RATMAC program as, say,

BINREAD: (⟨device⟩, ⟨parity⟩, ⟨buffer⟩, ⟨number of words⟩, ⟨parameter array⟩),

where the definition is

```
ANSI Fortran:  MACRO:(BINREAD:,READ($1)
                 ($3(KIK),KIK = 1,$4))
UNIVAC 11ø8:   MACRO:(BINREAD:,CALL AA88
                 ($1,$2,$3,$4,$5))
PE 8/32:       MACRO:(BINREAD:,CALLSYSIO
                 ($5,X'59',$1,$3,$4*4,ø))
CDC Cyber73:   MACRO:(BINREAD:,BUFFER
                 IN($1,$2)($3(1),$3($4));IF
                 (UNIT ($1).EQ.1)IQUIT = 1).
```

Macros may also call other macros. For example, a macro that limits a bit-string to the length of one word could be of the form

```
MACRO: (STRLIM:,$2 = MINø($1,BITSWORD:)).
```

It is also possible to perform arithmetic and logical operations within the macro itself and have the result deposited in the Fortran. A typical use might be in a macro that converts the number of words into the number of characters; for example,

```
MACRO:(CONVCHAR:,[$2 = arith:($1,*,arith:
    (BITSWORD:,/,BITSCHAR:))])
```

would mean that the RATMAC statement CONVCHAR:(15,NCHAR) would be processed into Fortran as NCHAR = 60 (using the values of BITSWORD: and BITSCHAR: defined above). The macro arith: does built-in integer arithmetic.

Macros are capable of referencing their own names in *recursive-type* operations. A good example of a recursive macro is given by Munn & Stewart (1978) for converting an octal number to decimal. The definition string is

```
MACRO: (LSTCHR:,[substr:($1,lenstr:($1),1)])
MACRO:
(FSTPART:,[substr:($1,1,arith:(lenstr:($1),–, 1))])
MACRO:(OCTAL:,[ifelse:($1. ø,[arith:(LSTCHR:
($1),+,arith:(8,*,OCTAL:(FSTPART:($1))))])]).
```

The RATFOR statements N = OCTAL:(177) and I = IA(OCTAL:(77),2) generate the Fortran statements, N = 127 and I = IA(63,2).

In summary, the combination of the RATFOR and MACRO features in the RATMAC preprocessor provides important benefits to a program system. Macros make it possible to use machine-specific functions in the system and yet to isolate them totally from the program code. The definitions of machine-specific macros are placed at the front of the system and are adjusted to the local computing environment before implementation. Apart from this no other changes to the system code should be necessary. In general, RATMAC code is more concise, logical, legible, transportable and adaptable than Fortran. Because machine-specific features can be used the resulting compiled code will also be more efficient. Further examples in the use of RATMAC will be discussed in the subsequent sections.

A more subtle advantage of using RATMAC is that the syntax of the language is under *user* control. The only limitation on the syntax is that the syntactical construct be translatable into ANSI Fortran. Thus, for example, if a CASE statement should prove to be a desirable programming construct, it can be added by the users, independent of whether ANSI Fortran allows such a construct. A concomitant advantage of this is that the user can be largely isolated from changes in the Fortran standard. Instead of each individual program being modified to conform to any new Fortran standard, only the preprocessor needs to be altered to generate Fortran compatible with the new standard.

3. The system structure

In the *Introduction* we discussed the importance of planning how the component parts of the program system are put together. Two fundamental considerations in system structure are (1) the micro-structure of the individual programs, and (2) the load-module structure of the system as a whole.

Microstructure

Microstructure is the term we shall use to describe the organization and partitioning of code *within* a calculation unit. There are many ways to approach this aspect of program design and each programmer tends to have a particular style based on past experience (you can always recognize the old machine-coders!). There are, however, two essentially distinct approaches. One is to partition a calculation into a large number of small subroutines, each with a specific task. The other is to write programs composed of essentially contiguous code and containing a minimum of calls to sub-routines.

Both approaches have their strong points. The first reduces redundancy of code and therefore minimizes core utilization. To many programmers it also simplifies writing and reading codes in a similar way to the RATMAC macros. However, unlike macros, subroutines usually remain partitioned at execution time. This means that there can be significant over-heads associated with the subroutine calling sequence and this can have detrimental effects on computational efficiency. The strength of the contiguous in-line code approach is the reverse. Large block programs contain

redundant code but are usually faster than their multi-subroutine counterparts.

However, even in this latter case RATMAC can offer significant advantages through a macro defined internal 'procedure' mechanism. Details of the three macros needed are given in Munn & Stewart (1979) together with examples of their use. 'Procedures' essentially remove redundant code by the use of ASSIGN'd GOTO's.

The XTAL system uses both approaches according to the circumstances. For instance, subroutine calls are avoided in the 'inner loops' of CPU-intensive calculations. The use of RATMAC macros often reduces the need for smaller subroutines. However, where significant savings are possible through specialized subroutines, they are used. This is the case for systems operations such as line and file input–output, memory allocation, bit-string manipulation and packing, and other service control functions. In the XTAL system, as in the case of the XRAY system (Stewart, 1976*a*), subroutines performing these tasks are grouped together and referred to as the *system nucleus*. The importance of the system nucleus concept to the XTAL microstructure is threefold. It eliminates redundancy for the most commonly performed operations; it concentrates in one place those subroutines which are most susceptible to implementation difficulties; and lastly, and most importantly, it represents the controlling subunit and communication link for the whole program system. Further details of the nucleus are given in § 4.

The implications of microstructure on a calculation are also dependent on the type of load module to be used at execution time.

Load-module structure

The finite size of direct-access memory usually requires that a program system be subdivided into convenient modules for loading. As discussed in § 1, there are at least three distinct methods of loading a program system – stand-alone, overlay and VMS paging. Unlike the XRAY system, which was intended mainly for use in overlay mode, the XTAL system structure takes into account both the stand-alone and overlay loading methods. Implementation is also straightforward on VMS machines, but the different methods of 'paging' make loading characteristics unpredictable. In theory, VMS should be most efficient when operations are concentrated in contiguous parts of the load module. The XTAL structure attempts to satisfy this requirement in major routines and in the way it stores and manipulates data (see § 6). However, there is some uncertainty in terms of overall VMS efficiency about the effect of repeated references to nucleus subroutines. One solution to this on some machines may be to force the nucleus subroutines to remain resident and not be paged. Fig. 1 shows diagrammatically how the different loading methods work with successive XTAL calculations.

An essential consideration in the load-module structure of a program system is the ability of modules to communicate data to each other. In the XTAL system data exchange takes place at three levels. The first is *via* a data file containing detailed archival information about the problem in hand. The second level is *via* the system common block (SYSCOM) which contains essential control parameter information. For the overlay and VMS execution procedures the SYSCOM remains intact for the duration of a complete XTAL run. However, in the stand-alone mode SYSCOM is stored and recovered automatically (on a scratch file) between each load (see Fig. 1). This facility permits the nucleus routines to perform their control functions quite independently of the loading procedure. The third level is *via* a common block for a given calculational segment.

4. The XTAL system nucleus

The concept of the XTAL system nucleus was introduced in § 3. The nucleus contains those subroutines that either perform operations common to all programs in the system or are necessary to control the interaction of these programs.

In the XTAL system nucleus subroutines have

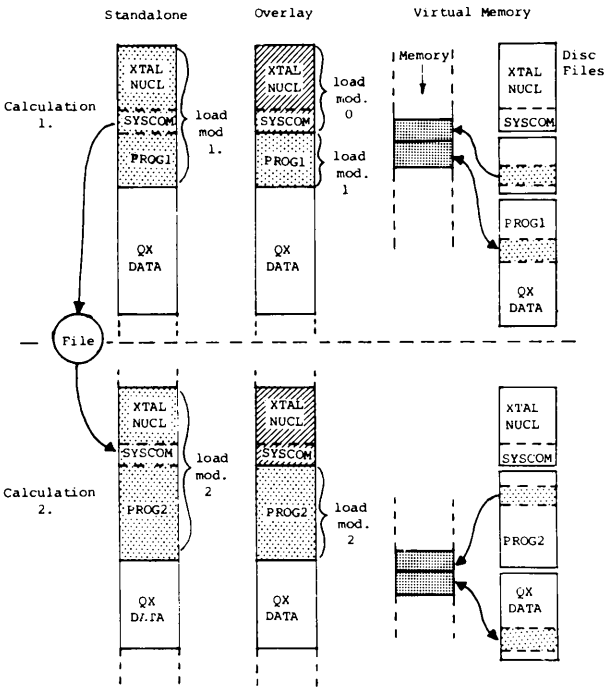

Fig. 1. XTAL loading modes.

names of the form 'AAnn' and are catalogued according to their function as follows:

AAØØ–AA19: system control, line input–output, and bit-string manipulation;
AA2Ø–AA39: binary sequential file input–output;
AA4Ø–AA59: memory allocation, data–time and error processing;
AA6Ø–AA79: direct-access file input–output;
AA8Ø–AA99: machine-specific subroutines which supersede or complement commonly used operations.

In practice the nucleus routines are referenced using descriptive macro names.

The number of nucleus subroutines is always kept to an absolute minimum and usually only subroutines AA8Ø–AA99 will vary from machine to machine. Here is a brief description of the existing XTAL nucleus subroutines for the machine-independent part of the system.

Macro	Subroutine	Description
—	AAAA	Main entry-point routine in the XTAL nucleus which is responsible for system (once-only) *start-up procedures*, the loading mode and SYSCOM storage and recovery.
—	AAØØ	Responsible for *program scheduling*, as directed by line input control parameters.
READLINE:	AAØ1	*Reads input lines* (images) and identifies command parameters (described in § 5).
DCODEFLD:	AAØ2	*Decodes input lines* into numeric and alpha field data (described in § 5).
MOVEWORD:	AAØ3	*Moves word-strings* from one location to another in the one-dimensional QX array.
MOVEBYTE:	AAØ4	*Moves byte-strings* from one location to another in the QX array.
MOVECTOR: MOVERTOC:	AAØ5	*Packs and unpacks byte-strings* into word strings, and *vice versa.*
COMPCHAR:	AAØ6	*Compares a character-string* with a series of character strings to find a match.
NCODEFLD:	AAØ8	*Outputs line* information (described in § 5).
WRITEPKT:	AA21	*Writes binary sequential data* file as packets in directory-type records (contents described in § 7).
READWPKT:	AA22	*Reads binary sequential data* file.
PKTPOINT:	AA23	*Extracts pointers* to specific data in packets from the record directory.
COPYFILE:	AA24	*Copies binary sequential data* file in two-buffer mode.
QXMEMORY:	AA41	*Allocates memory* by expanding or contracting the QX data array (described in § 6).
—	AA42	*Handles all error conditions* and exits *via* OZØØ or returns to calling program.

In addition to the machine-independent AA routines there may be some machine-dependent ones supplied; for example:

—	AA81	*CDC SPECIFIC* converts packed ASCII numeric characters to integers. Fortran routine used by the time and data routines.
	AA82	*CDC SPECIFIC* expands and releases memory to resident program. COMPASS routine used by AA41.
—	AA83	*CDC SPECIFIC* moves byte-strings from one location to another. COMPASS routine replaces AAØ4 in CDC version of the XTAL nucleus.
—	AA84	*CDC SPECIFIC* compares two byte-strings. COMPASS routine used by AAØ6.

There are three additional system nucleus routines which, in overlay mode, load as separate modules. This is because their function is connected with the start-up and shut-down of calculations and because they are only resident in memory during the transition from one calculation to the next.

MTØØ Calculates memory required for next load module and the area used in the QX array as

a program common block. MTØØ also reports the times and memory utilized in the previous program.

SYØØ System start-up initialization, and sets system common SYSCOM.

OZØØ Error exit routine; tells the user reason for premature shut-down.

5. Line input–output control

In the XTAL system, input or output images are not processed by Fortran I/O routines. The Fortran READ/DECODE/ENCODE/WRITE/FORMAT functions have been replaced by the XTAL nucleus subroutines AAØ1 (READLINE:)/AAØ2 (DCODEFLD:) and AAØ8 (WRITLINE:)/AAØ7 (NCODEFLD:). This has been done for three reasons: the nucleus subroutines are smaller than most Fortran library routines; they have features especially tailored to XTAL needs; and they avoid the difficulties often associated with the inconsistencies of Fortran I/O as implemented on various machines.

Line input

This is handled exclusively by routines AAØ1 and AAØ2. AAØ1 reads a line image of characters from the input file or device and places it in the buffer BFINIM. It then scans the leading characters of this buffer for an identification code which indicates whether the line image is for *system control*, *program initialization*, or *program parameter input*. If a system control code is detected AAØ1 uses AAØ2 to decode numeric information from the character buffer BFINIM into floating-point buffer BFINFP. The control function (see Table 3) is then completed before reading the next input image.

Table 3. *System input control functions*

Name	Function
TITLE	alpha-numeric page leader
REMARK	alpha-numeric line insert
FILES	I/O file assignments
MEMSET	preset memory allocation
SETID	preset input image names
FIELD	preset input field formats
ORDER	preset input field order
FINISH	shut down system and exit

If the identification code is not a system control code, the AAØ1 returns to its caller, which for program initialization codes will be the system scheduler AAØØ, and for program parameter codes will be the program itself. Again AAØ2 is used to convert numeric character information into floating-point numbers.

A powerful feature of the AAØ1/AAØ2 routines is the provision for either *free or fixed format* line input. The user may fix the input image format with the FIELD control image. In free-format mode, numeric or alpha-numeric data are delimited by a blank or a comma. There are also special codes $n and *n that enable skipping and positioning of data, and an added convenience for interactive input. The image

PARAMS 1,23,,5.72 $3 1ØØØ *3 77.−1 * 6 256

is decoded into successive fields containing 1.,23.,7.7,5.72,null,256.,null,1ØØØ.

Another feature of AAØ2 is its flexibility in decoding numeric data of different forms. For instance, each field in the image

3 3.Ø 3Ø.−1 +3 +3. +.3+1 +3Ø.−1 3Ø−1 .Ø3+2

would be decoded as the number three.

Line output

This is controlled by routines AAØ7 and AAØ8. AAØ7 is responsible for translating floating-point data contained in a specified input array into character strings in an output buffer. The output routine AAØ8 can automatically provide both page headings and sub-headings for column output if these are desired.

The output format for AAØ7 is specified by a coded string of the form CCCLLDT where CCC is the right-justified column of the character string, LL is the total length of the character string, D is the number of digits after the decimal point for E- and for F-type formats or the number of 'forced' digits for I-type format, and T is the format type. There are nine format types:

(1) E-type format S.YYYY±ZZ (S is blank or −);
(2) F-type format SXXX.YYY;
(3) I-type format (base-1Ø) SXXX (Ø < X < 9);
(4) I-type format (base-2) SXXXX (Ø < X < 1);
(5) I-type format (base-8) SXXXX (Ø < X < 7);
(6) I-type format (base-36) SXXXX (Ø < X < Z);
(7) binary packed number YYYY (Ø < Y < 1);
(8) octal packed number YYYY (Ø < Y < 8);
(9) hexadecimal packed number YYYY (Ø < Y < F).

Format specification is simplified for the programmer by the macro FMT: (CCC,QLL.D) for the common format types (T=1,2,3). QLL.D is the Fortran-type parameter where Q is an E for T=1, and F for T=2 and an I if T=3. An important feature of AAØ7 is its ability to change automatically the specified format to avoid an overflow of the LL parameter while maintaining the maximum number of significant places. If the number −1234.56789 was passed to AAØ7 with the format FMT: (1Ø,F10.5) or 1Ø1Ø52., it would be output as −1234.5679 (automatically dropping the last decimal place). If the format was FMT: (1Ø,F6.3) the number would be output as −1235. Yet again, if the format was FMT: (1Ø,F5.3) then the output number would automatically adopt an E-type format −.+Ø4. Further reduction in LL to 3 would result in a field overflow and

*** would be output. This illustrates the flexibility of AAØ7 in printing some information wherever possible.

When the numeric data have been formatted, the output buffer is written out by the routine AAØ8. AAØ8 is responsible for three functions: outputting numeric data and header information to both the primary and secondary line output files, controlling pagination and page title output, and *monitoring* line ouput *priorities*. All lines passed to AAØ8 are assigned an output priority, 1 to 5. Separate output *priority limits* are set for two separate output devices and this limit may be varied during an XTAL run. In this way the user can control the nature and the extent of line output on either device. The priority assigned to each line sent to AAØ8 is based on its importance to the user. Priority categories are:

line priority 1: essential output;
line priority 2: abbreviated output;
line priority 3: standard output;
line priority 4: extended output;
line priority 5: complete output, with I/O and memory diagnostics.

A typical use of this facility would be to set the priority limits of the output files to 2 and 4. In this way the user can list (or scan with a CRT) the abbreviated file containing lines of priority 1 and 2, and then decide if further details are required from the extended output file containing lines of priority 1, 2, 3 and 4. This type of output control is important for interactive use of the XTAL system.

6. Memory and data management

The importance of carefully planned data management in program systems was stressed in the *Introduction*. The size and diversity of crystallographic data, the frequent use of word-packing, and the range of machine types currently in use make this one of the most difficult aspects of system design. We shall cover this topic in four parts: word specification, system common, program common and the QX data array.

Word specification

This is critical to any system because of the variation in the length of a single word from machine to machine, and because of the different ways integer, floating-point (real) and character information is used within these words. The XTAL system is designed to accommodate machines with a minimum integer word length of 16 bits and a minimum real word of 32 bits. Allowing for 16-bit integers, with the implied maximum magnitude of 32 767, places the emphasis of data handling on real words.

Real words are used for packing so that a full 32 bits are available for this operation.

The only purpose for which integers must be used in XTAL is as indices to the QX data array. For machines with a 16-bit integer word the upper index limit of 32 767 is unlikely to prove a problem because of the commensurate limitation to the size of direct-access memory. Packed integer numbers (in a real word) are used extensively in order to reduce memory demand and increase computational speed. To ensure optimal speed for the packing/unpacking operations the macros MOVEBITS: and INTPACK:/INTUNPACK: are used. These macros are designed to use the fastest bit manipulation software available at each installation.

System common

This is the labelled COMMON/SYSCOM/ consisting of some 25Ø words. SYSCOM is the data communication region for all subroutines within the nucleus and for all programs to the nucleus. The actual contents of SYSCOM depend on machine-specific (macro) parameters which ensure that its length is kept to an absolute minimum. SYSCOM may be the only labelled common in XTAL depending on the definition of the macros COMI:, COMF:, COMC: (see below). This is a considerable departure from the XRAY system and arises principally from the requirement of some overlay loaders that *all* labelled commons must also be declared in the root element. Limiting labelled commons also reduces problems with VMS execution on machines where common blocks are addressed indirectly through page Ø.

Program common

For large calculations it is often necessary to have an efficient method for communicating data between the program subroutines or overlay segments. Traditionally this is done by using labelled commons specific to these routines. However, as discussed above, multiple use of labelled commons may not be efficient with some loaders. In the XTAL system flexibility is maintained by using the macros COMI:($1), COMF:($1) and COMC:($1) as the *labels* of any *local* labelled commons involving integer, real and character variables, respectively. $1 is the program name prefix. In this way, the XTAL installer can globally edit the local labelled commons to suit the requirements of the compiler and loader.

QX data array

This is the single one-dimensioned array used to store all directly addressable data in the XTAL system. In stand-alone and overlay loading modes, the length of the QX data array is varied dynamically according to the number of words currently needed to store data. For VMS operation, dynamic core allocation is

unnecessary and is disabled. Fig. 2 shows a typical layout of the QX data array. The array QX(1) is declared in SYSCOM and is assumed to be open-ended up to a possible memory maximum defined by MEMMAX:. Certain operating systems and loaders require that some memory immediately following SYSCOM be reserved for program code. Fig. 2 shows that the first 'usable' word of the QX array is defined as QX(QXSTAR+1). The value of the marker QXSTAR is non-zero for machines requiring that program code be loaded *after* SYSCOM. The value of QXSTAR needed to 'skip' the program code for each calculation is stored in the nucleus memory initialization routine MTØØ (see § 4).

MTØØ is also responsible for setting the marker QXWORK, which defines the last word of the QX array area which is the minimum data storage necessary for this program. The region QX(QXSTAR+1) to QX(QXWORK) is, in fact, a pseudo-common area which is *always allocated.* In dynamic allocation mode, the array is initialized at QX(QXWORK) and is never reduced below this point. Requests for additional QX memory are to the nucleus subroutine AA41, specified by the marker QXREQU. AA41 allocates QXREQU, if available, and returns the actual amount of memory allocated as the marker QXAVAL. It is *via* these two parameters, QXREQU and QXAVAL, that the QX array is lengthened or shortened according to current demand. At all times the user may override the dynamic allocation aspect of QX by entering a MEMSET parameter (see Table 3) to fix the length of the QX array.

7. Data file structure

The structure of the archive-type data files has in the past represented an obstacle to the use of program systems for many types of problems. This is because data files are usually of a *fixed sequential format.* Such a format provides for simple and fast access for the majority of problems but is relatively inflexible and even unusable in many situations.

The XTAL system uses a 'new' type of data file structure. As with the XRAY system, the file is divided into 'logical records' according to the type of crystallographic data. Each logical record is subdivided into units of information referred to as 'packets'. The first packet of each logical record contains the 'directory' to the contents of all succeeding packets in that record. In this way the packets and the logical records need only be as long as the data demand. The structure of the directory is simple. Each word in the directory packet contains an identification number which is unique to the specific crystallographic item stored in the identically ordered word in all subsequent packets of that logical record. Fig. 3 shows a typical data file structure. For the programmer, the task of extracting data from a directory-driven file is further simplified by the nucleus subroutine AA23 which points to the appropriate word in the packet containing a specified identification number.

It is worth noting that the logical record also contains information about the lengths of the packets, the number of packets, and the physical buffer lengths used in the input–output of the data file. However, this book-keeping information is of little interest to the XTAL user or, for that matter, to the XTAL programmer, because the physical aspects of file I/O are handled by the nucleus routines AA21–AA24. Details of the structure of the XTAL data file, and the assigned identification numbers, are given in Hall, Stewart, Norden, Munn & Freer (1980).

The concepts outlined in this paper have been the impetus for a cooperative programming effort sponsored by the National Resource for Computations in Chemistry (NRCC) and the National Science Foundation. The details of the proposed system are available in a number of technical reports which can be obtained from the University of Maryland Computer Science Center.

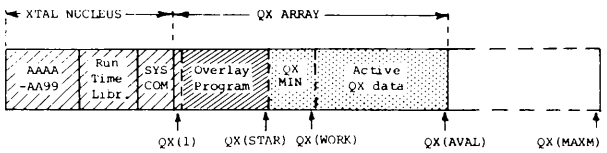

Fig. 2. XTAL data management.

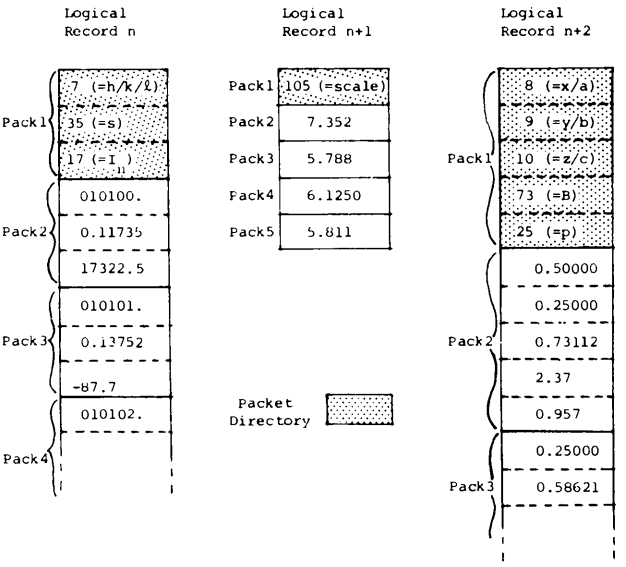

Fig. 3. XTAL data file structure.

The XTAL system and the concepts on which it is based have recently been used under the sponsorship of NRCC to develop a portable multiple isomorphous replacement program. A discussion of this project and its wider implication can be found in Robinson (1980).

References

HALL, S. R., STEWART, J. M., NORDEN, A., MUNN, R. J. & FREER, S. (1980). *The XTAL System of Crystallographic Programs: Programmer's Manual.* Report TR-873. Computer Science Center, Univ. of Maryland, College Park, Maryland.

KERNIGHAN, B. W. & PLAUGER, P. J. (1976). *Software Tools.* Reading, Mass.: Addison-Wesley.

MUNN, R. J. & STEWART, J. M. (1978). *RATMAC: Kernighan and Plauger's Structured Programming Language.* Report TR-675. Computer Science Center, Univ. of Maryland, College Park, Maryland.

MUNN, R. J. & STEWART, J. M. (1979). *RATMAC: A Primer.* Report TR-804. Computer Science Center, Univ. of Maryland, College Park, Maryland.

ROBINSON, A. (1980). *Science*, **207**, 746.

STEWART, J. M. (1976*a*). The XRAY system. Computer Science Center, Univ. of Maryland, College Park, Maryland.

STEWART, J. M. (1976*b*). *Crystallographic Computing Techniques*, pp. 433–443. Copenhagen: Munksgaard.

STEWART, J. M. & MUNN, R. J. (1978). In *Computing in Crystallography.* Delft Univ. Press.

Acta Cryst. (1980). A**36**, 989–996

On Formalism of Extinction Correction within the Validity Limits of the Mosaic Model

BY N. M. OLEKHNOVICH, V. L. MARKOVICH AND A. I. OLEKHNOVICH

Institute of Physics of Solids and Semiconductors, Byelorussian Academy of Sciences, Minsk 220726, *USSR*

(*Received* 14 *January* 1980; *accepted* 22 *May* 1980)

Abstract

The results of an investigation of the polarization coefficient of X-ray radiation diffracted in real crystals are given. The form of the angular dependence of the polarization coefficient in the range of the Bragg reflection is found to be qualitatively different in the cases of primary and secondary extinction. It allows the unambiguous identification of the type of extinction in the crystal. On the basis of the experimental data analysis of the polarization coefficients for silicon and germanium crystals with different dislocation densities, it is shown that the mosaic model of a crystal is suitable for describing X-ray scattering in real crystals if the dislocation density is higher than 10^4 mm^{-2} and in practice only primary extinction is present in mosaic crystals. An expression is given for the primary extinction factor for the mosaic crystal, obtained on the basis of the solution of the Takagi–Taupin equations for finite crystals. This expression was used for the analysis of the LiF and NaF structure factors measured by different authors. The effective size which was obtained for the domains appeared to be physically reasonable and to be directly connected with the value of the dislocation density in the crystal.

1. Introduction

Extinction in X-ray crystallography is described in most cases in terms of the Darwin (1914) mosaic block model. Zachariasen (1967) developed the formalism of the extinction theory on the basis of the Darwin energy transfer equations and applied it in the analysis of X-ray data for a number of substances (Zachariasen, 1968*a*,*b*). The Zachariasen theory greatly renewed interest in extinction. Coppens & Hamilton (1970) generalized the Zachariasen approach in the case of extinction anisotropy. It was established by many authors that the Zachariasen formalism significantly improved the agreement between the calculated and corrected-for-extinction experimental structure factors. This formalism was refined by Cooper & Rouse (1970) and more strictly reconsidered by Becker & Coppens (1974*a*, 1975). At the same time, different authors pointed out the shortcomings of the indicated formalism. Its main limitation is the kinematical approach (Werner, 1969), for it is based on the transfer differential equations, which take no account of coherence, since they involve only the intensities of the beams. This method does not appear to be suitable for correcting for severe primary