AN SQL INTERFACE FOR THE IFS/2 KNOWLEDGE-BASESERVER:RELEASE2.

S.H. Lavington, N.E.J. Dewhurst, A.J. Marsh, J.M. Emby and D.R. Thoen

Internal Report CSM-180
Release 1 issued December 1992
Release 2 issued May 1993

Department of Computer Science
University of Essex
Colchester CO4 3SQ
U.K.
Tel: 0206 872677
email: lavington@uk.ac.essex

AN SQL INTERFACE FOR THE IFS/2 KNOWLEDGE-BASE SERVER: RELEASE 2.

S H Lavington, N E J Dewhurst, A J Marsh, J M Emby and D R Thoen

Dept. of Computer Science, University of Essex, Colchester CO4 3SQ.

Abstract.

A version of the IFS/2, known as the IFS/Q, has been designed to give direct support to SQL programs running on a host computer. This Report gives detailed specifications of one external and two internal software and firmware interfaces which have been created for the IFS/Q. Release 2 differs from release 1 mainly in the Interface B details ~ (see Section 4). We have also tidied up the IFS/Q library procedures, which are described in a companion document - (see ref. 9); for convenience this is included as Appendix B to this report.

1. Introduction.

The Intelligent File Store, IFS/2, is an add-on unit which gives hardware support for knowledge-based systems (refs. 1, 2). The IFS/2 adopts an approach characterised as *active memory*, in which structures such as sets, relations and graphs are both stored and manipulated within the same hardware unit. The IFS/2 aims to provide a repertoire of generic whole-structure operations which lie at the heart of many database and Artificial Intelligence systems. Examples are: pattern-directed search, join, transitive closure. To the user, the IFS/2 appears as an active, persistent, shared-memory unit which can be attached via a standard SCSI channel to any host computer.

Internally, the IFS/2 employs a modularly-extensible array of SIMD- parallel search modules, controlled by transputers. The hardware is semantics-free, but the OCCAM firmware resident on the IFS/2's controlling transputers can be adapted to suit a variety of higher-level software interfaces. The prototype IFS/2 which has been built at Essex has 27 Mbytes of semiconductor associative (ie content-addressable) cache, backed by 2 Gbytes of associatively-accessed discs. The basic unit of storage is the tuple. Hence, the whole memory is referred to as the Associative Tuple Store (ATS).

A software simulator of the IFS/2 has been written (refs. 3, 4, 5) which incorporates a C procedural interface containing the following groups of primitives:

a)	Housekeeping (eg save, restore)	5 procedures
b)	Search, Insert, Delete	3 procedures
c)	Tuple Descriptor Management	3 procedures
d)	Lexical Token Conversion (for strings)	5 procedures
e)	Label Management (for efficient labelling of tuples)	5 procedures
f)	Relational Algebraic Operations (eg join)	7 procedures
g)	Graph Descriptor Management	3 procedures
h)	Graph Operations (eg transitive closure)	13 procedures.

The software simulator has been used at a number of sites to support higher-level information models. For example, it has been used to demonstrate efficient support for the CLIPS production-rule system (ref. 6). A version of the IFS/2 firmware has been developed which causes the actual hardware to implement most of the above procedural interface commands, except for those in groups (d), (e), and (g). This has allowed performance measurement and analysis to take place.

In carrying out the above work, it has become evident that an IFS/2 interface based on one-at-a-time operations is not always convenient. Although there is little *data* traffic between IFS/2 and host - (the unit buffers intermediate working sets) - overheads occur in transmitting the sequences of primitive operations which are necessary for the final execution of a composed, high-level language, task. Some means is required for packaging up related sequences of commands.

The packaging of commands requires a new interface to be designed. This interface could lie anywhere between the two extremes of either: (a) general-purpose and low-level, or (b) applications-specific and high-level. The former would require a 'machine- code' type of IFS/2 interface, which included constructs for iteration, etc. The latter implies a more declarative style. Bearing in mind that the main strengths of the IFS/2's architecture reside in the efficient execution of whole-structure operations rather than as a general-purpose computing engine, the declarative approach was felt to be more appropriate.

The next task was to select a suitable declarative high-level paradigm as the basis for the interface design. To retain some degree of generality, this should be language-specific rather than applications-specific. SQL was chosen as the starting-point. SQL has the advantages of DBMS comprehensiveness and commercial acceptability, but lacks expressiveness. However, there are several proposals for extending SQL (eg ESQL) so as to include deductive capabilities. Furthermore, there are examples of projects which are building AI-related systems on top of ESQL parallel platforms. Finally, there is a useful sub-set of SQL which has a simple syntax and is sufficient for running industry-standard query benchmarks such as AS^3AP.

In the rest of this report, we describe an SQL interface for the IFS/2. Implementing this interface has involved the design of software in two general sections: (a) written in C for the host computer; (b) written in OCCAM for the attached IFS/2 unit. In order to distinguish this version of the OCCAM firmware from other versions, the IFS/2 which supports SQL is known as the IFS/Q. The IFS/Q development has required three identifiable software interfaces to be defined. The first constitutes the particular sub-set of standard SQL which is supported by IFS/Q in Release 2. This is obviously of interest to end-users. The other two interfaces are internal, and of primary interest to system developers and members of the IFS/Q development team.

This Report concentrates on the internal interfaces. The IFS/Q host procedures have been made into a C library, which is described in Appendix B. This library provides a general platform upon which SQL-oriented information systems may be built.

2. Overview of the IFS/Q.

2.1 Information flow

The IFS/Q system appears to the user as a normal UNIX Workstation with an extra 'desk-side' unit connected to one of its SCSI ports. Figure 1 shows this in more detail, in which a Sun Workstation is the host. The IFS/2 hardware unit is connected to this via a standard SCSI channel. Software written in C and running on the Sun is responsible for providing a convenient SQL programming environment, including the reporting of syntax errors and displaying the final results. The IFS/2 add-on unit is responsible for storing and processing data held as relations, and for carrying out certain memory-management tasks appropriate for a database system. Firmware written in OCCAM organises the IFS/2's SIMD search modules so as to carry out hardware-assisted relational algebraic operations, in response to SQL statements.

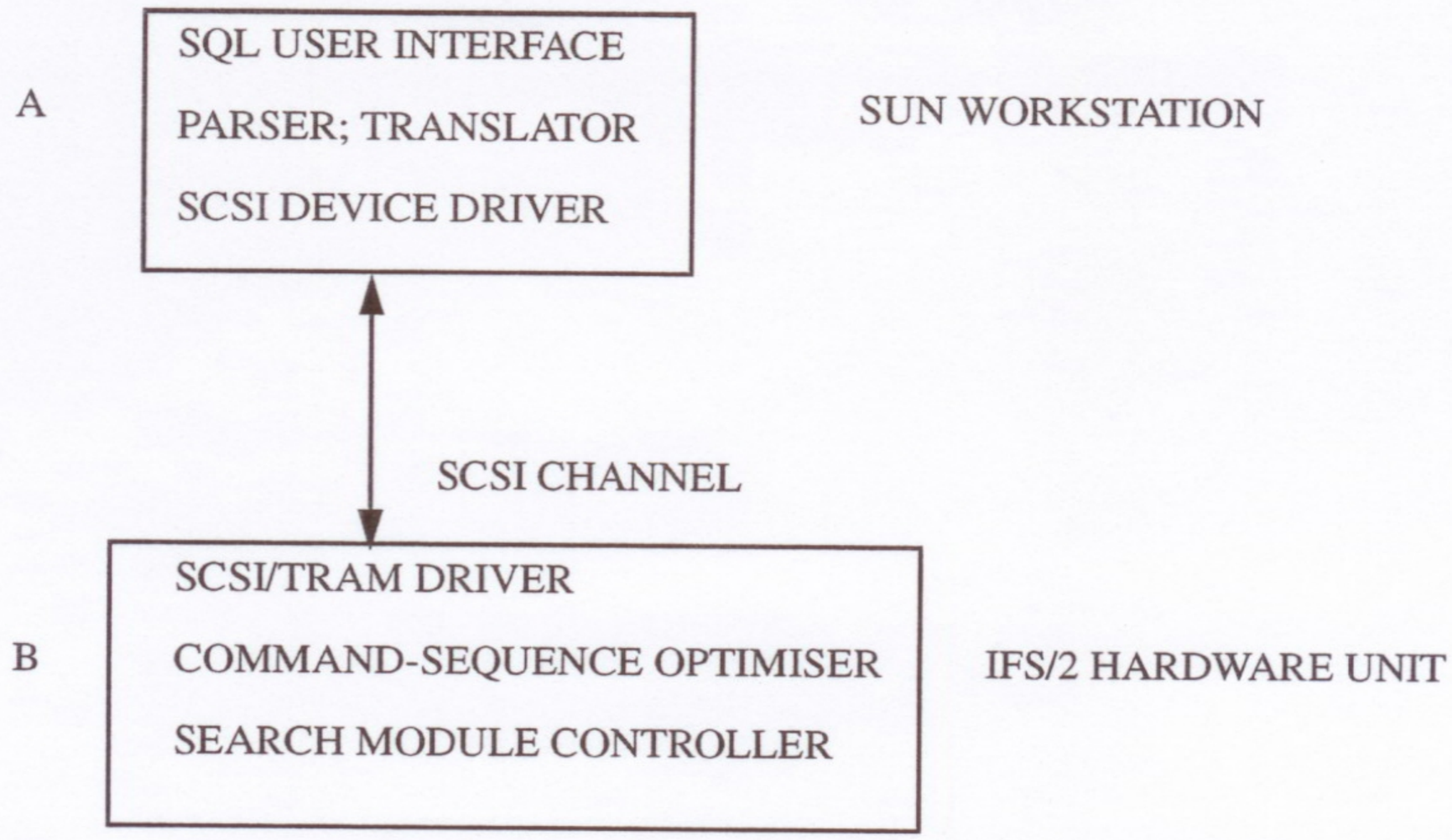


Figure 1. Overall system diagram of the IFS/Q, showing principal software and firmware modules.

In Figure 1, a user types in an SQL statement which constitutes a logical unit of work. In general, the SQL statement specifies a group of actions to be performed on given relations which may transform the database from one consistent state into another consistent state. The 'logical unit of work' may for example cause results to be returned to the user, or cause a new relation to be created in the database.

In Figure 1 the parser checks that the SQL statement is syntactically correct. The translator in Figure 1 interprets SQL-specified actions and produces a tree of corresponding IFS/2 relational algebraic commands. At this stage, some algebraic optimisations are performed which are independent of the IFS/2 architecture.

The relational algebraic tree is passed across the SCSI interface to the IFS/2 unit. The command-sequence optimiser in Figure 1 takes this tree and produces a linear sequence of low-level IFS/2 commands which are defined and optimised in terms of an Abstract IFS/2. (This Abstract Machine does not need to contain notions of parallelism, or knowledge of physical configuration parameters such as number of IFS/2 nodes or number of search modules per node). The search module controller in Figure 1 interprets this Abstract Machine sequence as a series of hardware actions involving the SIMD-parallel IFS/2 hardware. The controller allocates physical resources dynamically, to suit the package of relational algebraic work being undertaken. It also looks after low-level, hardware- related, activities such as inter-node communication.

Note that Figure 1 is a gross simplification of the IFS/2's internal architecture. In particular, the 'search module controller' consists in reality of many OCCAM processes running on several transputers.

As far as the specification of software is concerned, there are three important interfaces inherent in Figure 1:

SQL dialect: the repertoire of SQL commands available to external users. This is discussed further in Section 2.2.

Host/Abstract IFS: the relational algebraic tree representation of an SQL statement (at point A in Figure 1); also, the means for passing results back to the SQL user. See Section 3 for details.

Abstract/hardware IFS: the optimised sequence of commands for the Abstract IFS/2 (at point B in Figure 1); also, the means for passing results back for eventual presentation to the user. See Section 4 for details.

2.2 IFS/Q dialect of SQL.

The first version of IFS/Q supports a useful sub-set of standard SQL. It is not the intention at this stage to provide a full DBMS along the lines of, say, INGRES, so facilities concerned with database views, integrity, recovery, etc., are absent. On the other hand, the IFS/2 is essentially an associative (ie content-addressable) memory so that the full power of the relational model can be explored without the user having to worry too much about storage structures and their effect on run-times during query processing. Technicalities such as key (index) fields and access methods are largely irrelevant. Thus, INGRES commands such as MODIFY, CREATE_INDEX, etc. are also irrelevant and are not supported by IFS/Q.

The full BNF syntax for the first version of IFS/Q SQL commands is given in Appendix A. The main omissions at this stage are seen to be the ORDER_BY, GROUP_BY, and HAVING clauses within the SQL SELECT statement, and the UPDATE statement. All of these omitted activities can of course be achieved indirectly, though at some inconvenience to the user. For example, an SQL UPDATE has to be achieved via a DELETE followed by an INSERT.

It will also be seen from Appendix A that the allowable data formats in IFS/Q are somewhat limited at present. In the first version of the IFS/Q, all fields in a table are stored as 32-bit

integers. Character strings, of up to 120 ASCII characters in length, are converted to unique 32-bit Internal Identifiers before being stored - using a software simulation of the IFS/1's Lexical Token Converter (LTC, ref. 7) written in C and running on the host. (The LTC will eventually be incorporated into the IFS/2's hardware). Numeric constants are all 32-bit integers.

In summary, the following types of SQL statement are supported in the first version of IFS/Q:

CREATE_TABLE	(used to create a relation)
DROP_TABLE	(used to delete a relation)
INSERT_INTO	(used to enter one or more tuples)
DELETE_FROM	(used to remove one or more tuples)
SELECT	(the main SQL data-manipulation command).

The usual arithmetic functions (+, -, *, / and modulo) and the usual aggegate or 'set' functions (avg, count, sum, max, min) can appear in the SQL SELECT and WHERE clauses. However, arithmetic exponentiation and the SQL partial-match functions (ie wild cards in character strings) are not implemented in the first version.

Users logged on to the host Sun Workstation will issue SQL statement(s), separated by semicolons, as UNIX command line(s). As usual, input is either from the keyboard or from a UNIX file. Facilities for bulk loading of tables, handling database schema, etc., are provided by the IFS/Q library - (see Appendix B). The first version of IFS/Q will handle databases of up to about 10 Mbytes - a limit set by the 27 Mbytes of semiconductor Associative Tuple Store in the IFS/2 prototype; (this cache is also used for relational algebraic manipulations). Extensions to the IFS/2's 2 Gbytes of associatively- accessed disc storage via semantic caching (ref. 8) may be introduced in later versions.

3. Host/IFS interface.

3.1 Overview.

As explained earlier, IFS/Q handles SQL statements via a new interface known for simplicity as Interface A - (see Figure 1). This is based on a package of work known as a transaction. Interface A at present defines seven types of transaction. Some of the key-words used to describe these IFS/Q transactions are naturally similar to standard SQL statements. However, it is important to remember the different contexts in which IFS/Q transaction-types and SQL statements appear. In particular, it is intended eventually to parameterise Interface A transactions - (see later). The seven types of IFS/Q transaction are:

type 0:	QUERY	(supports an SQL SELECT statement; responders, if any, are returned to the host)
type 1:	INSERT_INTO	(supports the general SQL INSERT_INTO statement, which may derive several new tuples from sub-queries on one or more existing tables)
type 2:	DELETE_FROM	(supports the general SQL DELETE_FROM statement, which may delete several tuples whose

specification is derived from sub-queries on one or more existing relations)

type 3:	INSERT	(supports the SQL INSERT_INTO statement where the field-values of a single tuple are specified directly)
type 4:	DELETE	(supports the SQL DELETE_FROM statement where the field-values of one or more tuples are specified or implied directly)
type 5:	CREATE_TABLE	(supports the SQL CREATE_TABLE statement; causes a new IFS/2 class to be declared)
type 6:	DROP_TABLE	(supports the SQL DROP_TABLE statement; causes an IFS/2 class to be undeclared).

The first three transactions perform most of the data-manipulation, via queries or sub-queries on tables. In the IFS/Q, an SQL table is represented by an IFS class (ref. 1). Queries or sub-queries are expressed as a binary tree, where each node specifies an IFS/Q relational algebraic primitive operation on one or more classes. The list of primitives, together with their Interface A operation-codes, is:

5	SEARCH	(this is the equivalent of relational select and project, carried out as a single IFS action)
11	JOIN	
12	UNION	
13	DIVIDE	
14	EXTEND_BY	(adds a new field to a relation which is a function of existing fields in that relation)
16	EXTEND_BY_AGG	(performs an aggregate calculation, and then extends each tuple by that value)

The above IFS/Q primitives each have multiple parameters, as explained more fully via BNF definitions in Section 3.2. SQL users wishing for a quick overview should now skip to the illustrative examples given in Section 5.

SQL statements are compiled into IFS/Q transactions. These are passed down the SCSI channel to the actual IFS/2 add-on unit as packets. Results are returned to the host as blocks of data. It is anticipated that the IFS/Q transactions may originate either as source statements, as at present, or from another applications program running on the host computer. In the latter case, it may be convenient to communicate with the IFS/2 in terms not of SQL source but in terms of lower-level 'pre-compiled' transactions, where queries are expressed as C structures representing relational algebraic expressions. Although not implemented yet, this route is indicated at point X in Figure 2. This Figure shows the general flow of information during the execution of an SQL statement. The relationship between Figure 2 and the suite of 22 C library procedures is explained in Appendix B. Figure 2 also shows, for completeness, the route taken by systems utilities such as IFS engineering test programs.

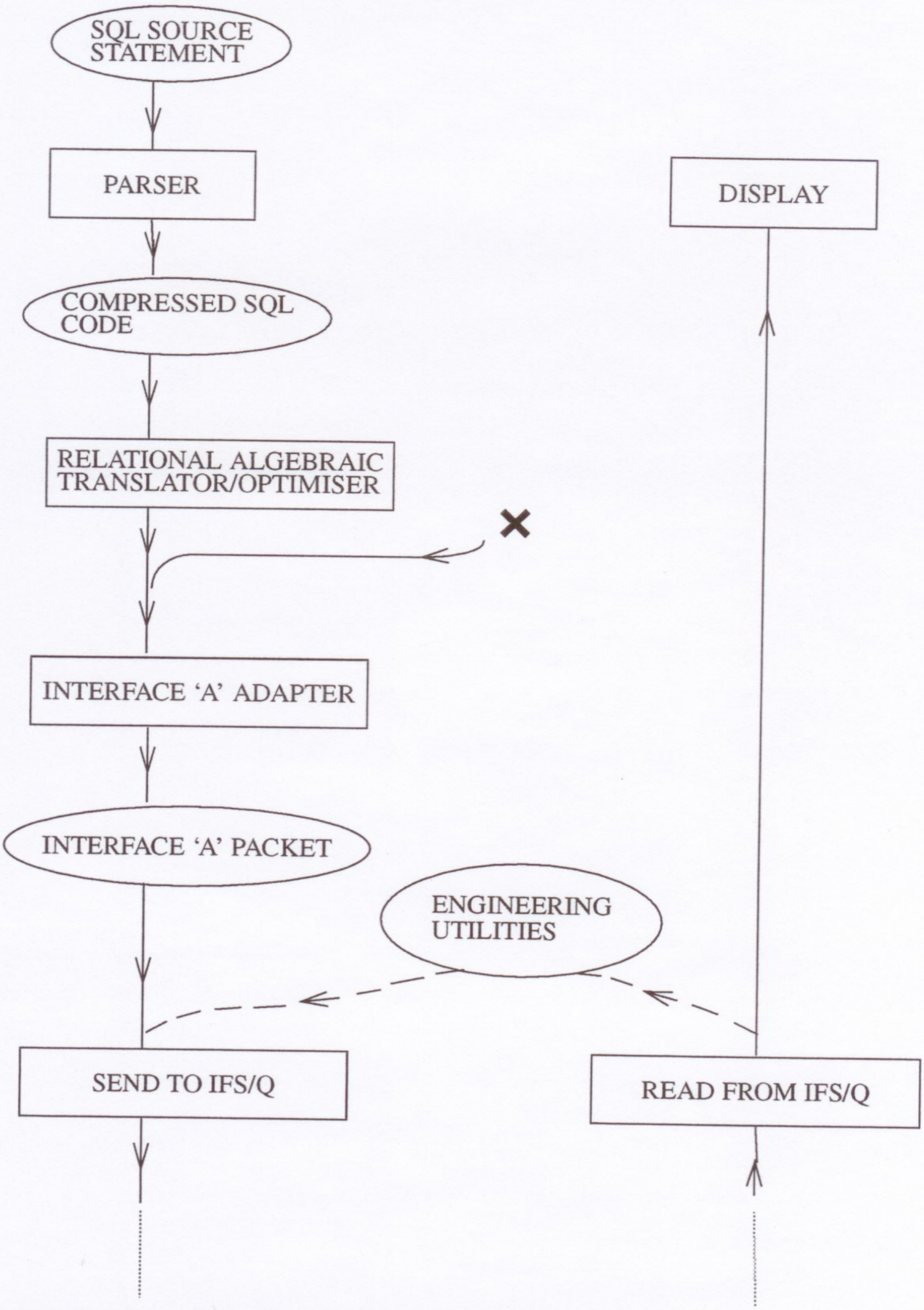


Figure 2: the modular structure of the host software.

3.2 BNF definition of Interface A.

We use a modified BNF notation, with the following conventions:

x y	means	x then y
[x y]	means	x or y, but not both
x | y	means	x or y, but not both
x[n]	means	x n times
x+	means	x at least once.

Comment lines begin with /*. Note that relations (ie SQL tables) are stored in the IFS/2 as classes. Thus, a *class_number* is used to identify a relation (ie table). Note also that numeric tokens or codes are used in Interface A for primitive operations, logical and arithmetic operators, etc. Each transaction begins with a 32-bit word *tr_length* which gives the total length in words of the string which follows - (see the examples in Section 5).

```
TRANSACTION ::=  tr_length  0  QUERY
                | tr_length  [1 2]  target_class  QUERY
                | tr_length  [3 4]  target_class  n  field[n]
                | tr_length  5  no_of_fields
                | tr_length  6  class_no

 /* Transaction types:
     0 = QUERY (return responders to host)
     1 = INSERT_INTO
     2 = DELETE_FROM
     3 = INSERT (a single tuple)
     4 = DELETE (   "     " )
     5 = CREATE_TABLE
     6 = DROP_TABLE
 */

QUERY   ::=  NODE+

NODE   ::=  node_id  ref_count  leftson_id  rightson_id  IFSQ_OP
```

```
IFSQ_OP  ::= 5 class_no  m  const[m]  op[m]  n  range_upper[n]  COND  RETURN
           | 11  COND  RETURN
               classno1  m1  const1[m]  op1[m]  n1  range_upper[n1]  COND
               classno2  m2  const2[m]  op2[m]  n2  range_upper[n2]  COND
           | 12  classno1  classno2  RETURN
           | 13  classno1  classno2  RETURN
           | 14  ARITH_EXP  RETURN
           | 16  [0 2 3 4]  field  RETURN
           | 16  1 RETURN

/* Tokens for IFSQ primitives:
    5   =  SEARCH
    11  =  JOIN
    12  =  UNION
    13  =  DIVIDE
    14  =  EXTEND_BY
    16  =  EXTEND_BY_AGG

   Tokens for set (aggregate) functions:
    0   =  COUNT
    1   =  SUM
    2   =  MAX
    3   =  MIN
    4   =  AVG
*/

   Codes for the 'op' array:
    0   =  WILD
    2   =  EQ
    8   =  NE
    4   =  GT
    6   =  LT
    10  =  IN_RANGE
    20  =  J_EQ
*/
```

```
COND        ::=  cond_length  n  CONJUNCT[n]
CONJUNCT    ::=  n  COMP[n]
COMP        ::=  [2 4 6 8 ]  ATOM  ATOM
  */   2 = EQ,  4 = GT,  etc,  as above  */
ATOM        ::=  0 const | 1 field
ARITH       ::=  arith_length  ARITH_EXP
ARITH_EXP   ::=  [60 61 62 63 64 ]  ARITH_EXP  ARITH_EXP
                 | 65  ARITH_EXP  |  ATOM

   /*  Tokens for arithmetic ops:
       60   =   ADD
       61   =   SUB
       62   =   MUL
       63   =   DIV
       64   =   MOD
       65   =   UMI(nus)
   */

RETURN    ::= n return_field[n]
```

4. Abstract IFS to hardware IFS interface

4.1 Overview.

This interface sits at point B in Figure 1; hence, it is known for short as 'interface B'. The command-sequence optimiser in Figure 1 takes the encoded query-tree as passed down from the host computer and converts this into a linear sequence of hardware-related IFS tasks. There are two main transformations carried out in moving from interface A format to interface B format: (i) query-tree node information is factored out; (ii) relational algebraic buffer information is inserted.

The concept of relational algebraic buffers, or RAP buffers, needs further explanation. The general form of an IFS dyadic operation on two relational structures is as follows:

(i) Select the first relational operand from the IFS's main associative memory, and load it into an associative buffer;

(ii) Select the second operand and load it into a second buffer;

(iii) Perform the specified relational algebraic operation (eg JOIN) between the first two buffers, placing the result in a third associative buffer.

In this respect, the sequence is analagous to register-loads followed by register-to-register arithmetic in a conventional computer. By further analogy, the IFS may need to assign several RAP buffers when evaluating a complex relational algebraic expression. At the level of the command- sequence optimiser (CSO) in Figure 1, RAP buffers are given logical numbers; at the lowest firmware level, these 'logical' RAP buffers are translated into real physical sectors of the IFS/2's SIMD associative memory. There is clearly a limit to the available associative memory, but experience suggests that a modest number of buffers is sufficient to implement most common SQL transactions. In the initial IFS/Q implementation, any query requiring more than 10 buffers is rejected by the CSO.

The Command Sequence Optimiser (CSO) in Figure 1 is written in OCCAM and runs on what is known as the TOP transputer within the IFS/2. Traffic across Interface B flows in both directions. Specifically, it is helpful to imagine that the TOP transputer can generate three (related) types of message:

(a) information about an SQL transaction, passed 'down' to the search module controller(SMC) as a sequence of tasks;

(b) result/status information, originating in the SMC but relayed via the TOP transputer back to the host Workstation;

(c) result/status information which originates in the Command Sequence Optimiser (CSO) and is sent back to the host.

Most IFS/Q transactions cause messages of types (a) and (b) to be generated. However, the CREATE_TABLE transaction only causes a message of type (c) to be generated because the IFS/2's Tuple Descriptor Table is stored and administered by the TOP transputer - (see later). We first concentrate on descriptions of type (a) messages, since these are the most complex.

The Command Sequence Optimiser (CSO) deals with the seven types of transaction described in Section 3.1, namely: type 0 (QUERY) through to type 6 (DROP_TABLE). For types 0, 1, and 2 the CSO works out the total number of logical RAP buffers required to carry out the transaction and sends this as a parameter, denoted in Section 4.2 as *'no_of_raps'*, in a special *create_rap_buffers* task. The special RAP buffer 0 is only used by individual search operations, and has the effect of sending results straight back to the host. More generally, if the host wishes to receive the results of a dyadic relational operation the command SHOW_RAP must be used - (see Section 4.2.1). If, on the other hand, the results of any relational operation are required to be inserted into, or deleted from, the ATS then the INSERT_INTO or DELETE_FROM commands should be used.

For transactions of type 0, 1, and 2 the main work is carried out by a sequence of one or more tasks which includes the Interface A operation- codes given in Section 3.1, namely: 5 = SEARCH; 11 = JOIN; 12 = UNION; ... etc. For these operations, further IFS/2-specific information is provided, as follows.

(a) Most dyadic operations take their inputs from two source-buffers and return the result to a third buffer. These three RAP buffers are identified in interface B by logical numbers, passed as three parameters known as: *rap_left*, *rap_right*, and *rap_result.* For the monadic SEARCH operation, the result is placed in a buffer given by the value of the

parameter *rap*. The full BNF definitions for each operation are given in Section 4.2 below.

(b) A buffer is usually cleared before it is used. In some cases (eg UNION) the initial contents of a buffer should be preserved; when this is the case, the *clear_flag* parameter is set to zero rather than one.

(c) When a relation or part-relation is transferred from the IFS/2's main associative memory to a RAP buffer, the order of the fields may be changed so as to give prominence to those fields taking an active part in subsequent relational operations - e.g. JOIN. Two interface B parameters control such interchanges. Firstly, the array parameter *return_fields[r]* gives the new ordering of fields. Secondly, the parameter *intersect_fields* indicates how many of these re-arranged fields, counting from the left, are of significance (ie 'active') in the parent relational operation.

One final point about housekeeping should be mentioned. All transactions involve base relations. Each relation known to the IFS/Q is given a class-number. Complete information about each relation (eg format, names of attributes, etc.) is kept in the Sun Workstation. A sub-set of this information is kept in the TOP transputer, in the form of a Tuple Descriptor Table (TDT). The TDT also keeps up-to-date information on relation cardinality, for use in optimising IFS tasks. The TOP transputer (running the CSO firmware) and the lower transputers (running the SMC software) communicate via a 'private' channel in order to keep the TDS up-to-date, etc. These housekeeping messages are not described in this Report.

As mentioned previously, an IFS/Q transaction is split into separate tasks for transmission 'downwards' to the SMC. Since the initial IFS/Q implementation is single-user, the tasks can be transmitted independently (but in sequence, naturally). The modified BNF definitions for interface B tasks are now given. Readers wishing for a quick overview should skip to the illustrative examples for both interface A and interface B, as provided in Section 5.

4.2 BNF definition of Interface B.

4.2.1 The principal messages sent from CSO to SMC

The following conventions are used.

x y means x then y
x | y means x or y, but not both
x[n] means x n times
x+ means x at least once

Comment lines begin with /*.

We first define the format of type (a) messages - (see Section 4.1) - ie, those that pass from the CSO down to the SMC. The codes (or tokens) used for transaction-type, IFSQ-primitive, set (or aggregate) function, and logical operation, are the same as those given in Section 3.2.

TASK ::= message_ length [3 4] INSDEL-INFO

| message_length	5	SEARCH_INFO
| message_length	11	JOIN_INFO
| message_length	12	UNION_INFO
| message_length	13	DIVIDE_INFO
| message_length	14	EXTEND_BY_INFO
| message_length	16	EXTEND_BY_AGG_INFO
| message_length	20	ASSIGN_RAP_BUFFERS_INFO
| message_length	21	SHOW_RAP_INFO
| message_length	22	INSERT_INTO_INFO
| message_length	23	DELETE_FROM_INFO

/*3 = insert;4 = delete; 5 = search; 11 = join; 12 = union, etc.
/* 20 = create RAP buffers
/* 21 = show RAP (ie return the contents of RAP buffer to the host)
/* 22 = insert contents of RAP into target_class
/* 23 = delete contents of RAP from target_class
/* N.B. - tasks 12, 13, 14 and 16 have not yet been implemented.

INSDEL_INFO ::= class_no m const[m]
/* m = no. of fields in the tuple.

SEARCH_INFO ::= m const[m] op[m] n range_upper[n]
COND RETURN class_no no_of_hash_fields RAP clear_flag

JOIN_INFO ::= RAP_left RAP_right RAP_result fields_left
fields_right no_of_intersect_fields no_of_hash_fields
COND RETURN clear_flag

UNION_INFO ::= RAP_source RAP_dest m no_of_hash_fields
RETURN clear_flag

ASSIGN_RAP_BUFFERS_INFO ::= no_of_RAPs

SHOW_RAP-INFO ::= no_of_fields RAP_no

INSERT_INTO-INFO ::= no_of_fields RAP_no target_class

DELETE_FROM_INFO ::= no_of_fields RAP_no target_class

```
/* rap, rap_left, rap_right, rap_result, are the logical numbers for
/* RAP buffers. intersect_fields gives the number of intersecting
/* fields.

COND        ::=  cond_length  n  CONJUNCT[n]
CONJUNCT    ::=  n  COMP[n]
COMP        ::=  [2 4 6 8 ]  ATOM  ATOM
  */   2 = EQ,  4 = GT, etc, as above  */
ATOM        ::=  0 const | 1 field
ARITH       ::=  arith_length  ARITH_EXP
ARITH_EXP   ::=  [60 61 62 63 64 ]  ARITH_EXP  ARITH_EXP
                 | 65  ARITH_EXP  |  ATOM

  /*  Tokens for arithmetic ops:
      60   =   ADD
      61   =   SUB
      62   =   MUL
      63   =   DIV
      64   =   MOD
      65   =   UMI(nus)
  */

RETURN    ::= n return_field[n]
```

4.2.2 Information returned from SMC to CSO, and from CSO to host.

These are the type (b) and type (c) messages that convey results and status information, as defined in Section 4.1. All except the CREATE_TABLE case originate at the SCM, though the host software does not need to be aware of this difference.

Return messages are packaged into 1024-byte blocks and sent over the SCSI interface to the host. The first eight bytes of each block form two 32-bit control words with the following format; these are followed by the actual data (if any):

first word: a count in the range 0 to 254 of the valid words of data in this transfer;

second word: bit 0 (least sig.) = 0 if this is the last SCSI block to be transferred;
= 1 if there are more blocks to come.

bit 1 = 0 if the command's <status> is successful;
= 1 if the command's <status> is not successful.

bits 2 - 30: unassigned at present.

bit 31 (most sig.) = 0 if the command's execution is OK;
= 1 if there has been a system error.

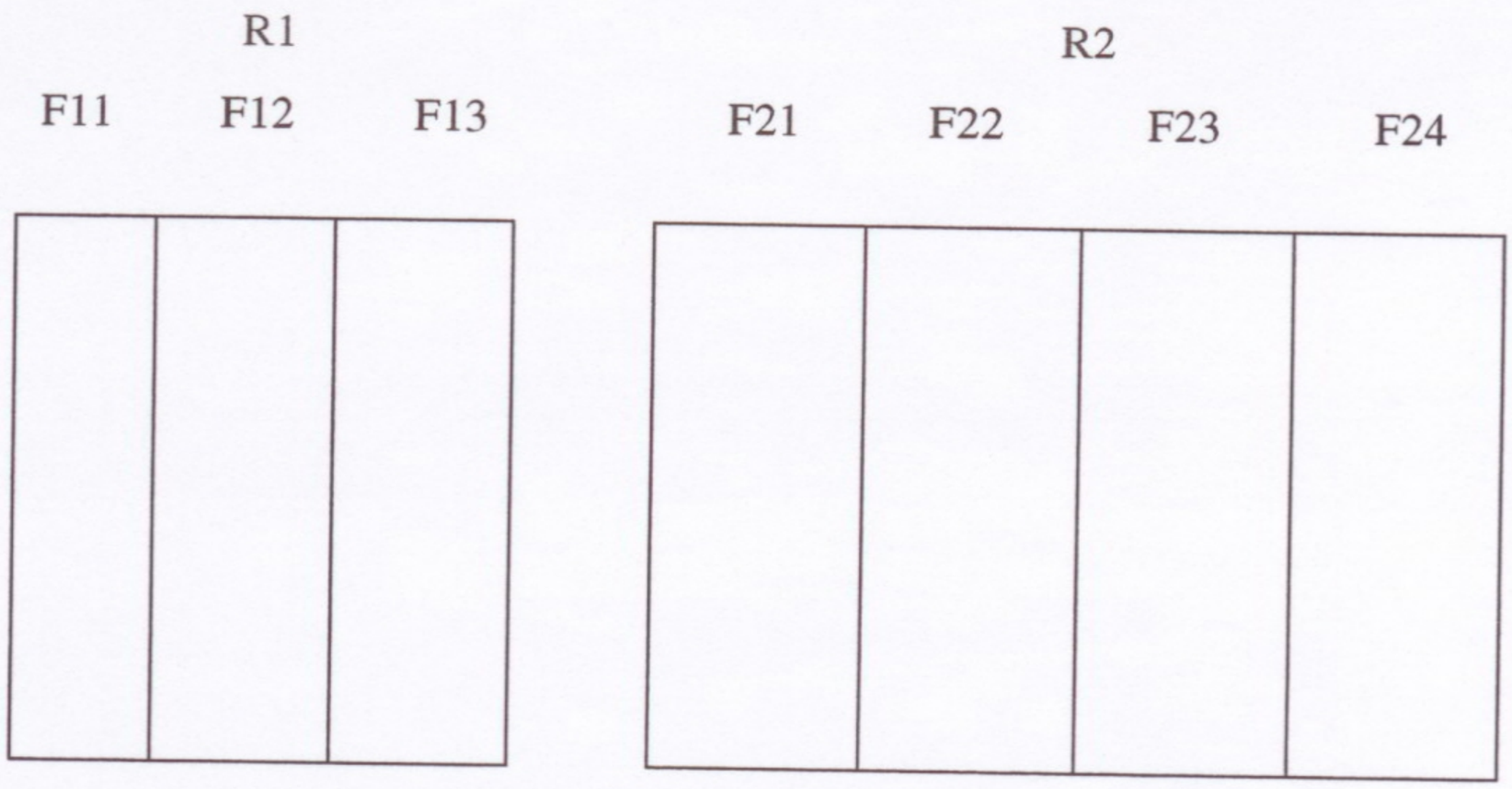
The remaining 254 words in the block may contain valid data, or (in the case of commands returning no data) undefined data, or (in the case of errors) diagnostic information. In the case of valid data, the host software already has SQL command information which enables the format of responders to be determined. Similarly, the interpretation of bit 1 is dependent on the SQL command previously sent.

In the case of system errors, it is intended that the CSO will return certain diagnostic information to the host. The format of this has not yet been finalised. An example of a system software error is the case where the host issues an SQL command which refers to a class-number which does not yet exist as far as the CSO is concerned. The CSO will also attempt to return meaningful diagnostic information in the case of an IFS hardware error at the SCM level. The SCSI channel's host device driver has a time-out arrangement, which will assist in cases of more serious hardware faults.

5. Example queries and their translation.

Suppose for the sake of illustration that a user had created two relations R1, R2. Suppose that R1 had three fields (attributes), known respectively as F11, F12, F13, and that R2 had four fields known as F21, F22, F23, F24.

Pictorially:



Suppose that, after being created, these relations had IFS class-numbers equal to 1 and 2 respectively.

Example query (a).

In English: "Retrieve F11 and F13 from all tuples in R1 having F12 equal to the constant + 23".

As an SQL statement:

```
SELECT F11, F13
FROM R1
WHERE F12 = 23;
```

This translates into an IFS/Q type 0 (QUERY) transaction, of length 19 words, having a single SEARCH node in its relational algebraic tree. This is represented at Interface A (Figure 1) as a string of 32-bit words. (A later version of IFS/Q may compress this to bytes where appropriate). At present, the string of words as decimal values is:

```
20   (transaction length, equal to20 words after this one)
0    (transaction type)
1    (node_id, for the single node)
0    (this node has a zero reference-count)
0    (leftson_id; in this case, there is no left son)
0    (rightson_id; in this case, there is no right son)
5    (the code for the SEARCH primitive operation)
1    (class-number of relation R1)
3    (information for three fields follows)
0    (there is no 32-bit constant relevant to field F11)
23   (the constant + 23, used in F12 matching)
0    (there is no constant relevant to field F13)
0    (a wild card is required when searching field F11)
2    (the code for equality matching, used when searching field F12)
0    (a wild card is required for field F13)
0    (no range information is required)
1    (condition length equal to one word after this one)
0    (no condition information is called for)
2    (return two fields, identified as follows)
0    (the first field of the tuple; fields (or columns) are numbered from zero)
2    (the third field of the tuple).
```

This is transformed into two tasks, to be sent by the Command Sequence Optimiser via Interface B to the Search Module Controller. The first task may be represented symbolically as:

2 :: [20 1]

This task is transmitted to the SMC as three 32-bit words:

2	message length
20	AllocateRAPs
1	one RAP needed

The second task is represented symbolically as:

18 :: [5 3 0 23 0 0 2 0 0 0 2 1 0 2 1 3 0 0]

It is transmitted as 18 32-bit words as follows:

18	message length
5	search
3	no of fields
0	wild field
23	constant field with value 23
0	wild field
0	treat field as wild
2	perform constant comparison on field
0	treat field as wild
0	zero upper limit ranges
1	condition length
0	zero conditions
2	two return fields

0 first field
2 third field
1 class no
3 dummy value
0 return data to host
0 dummy value

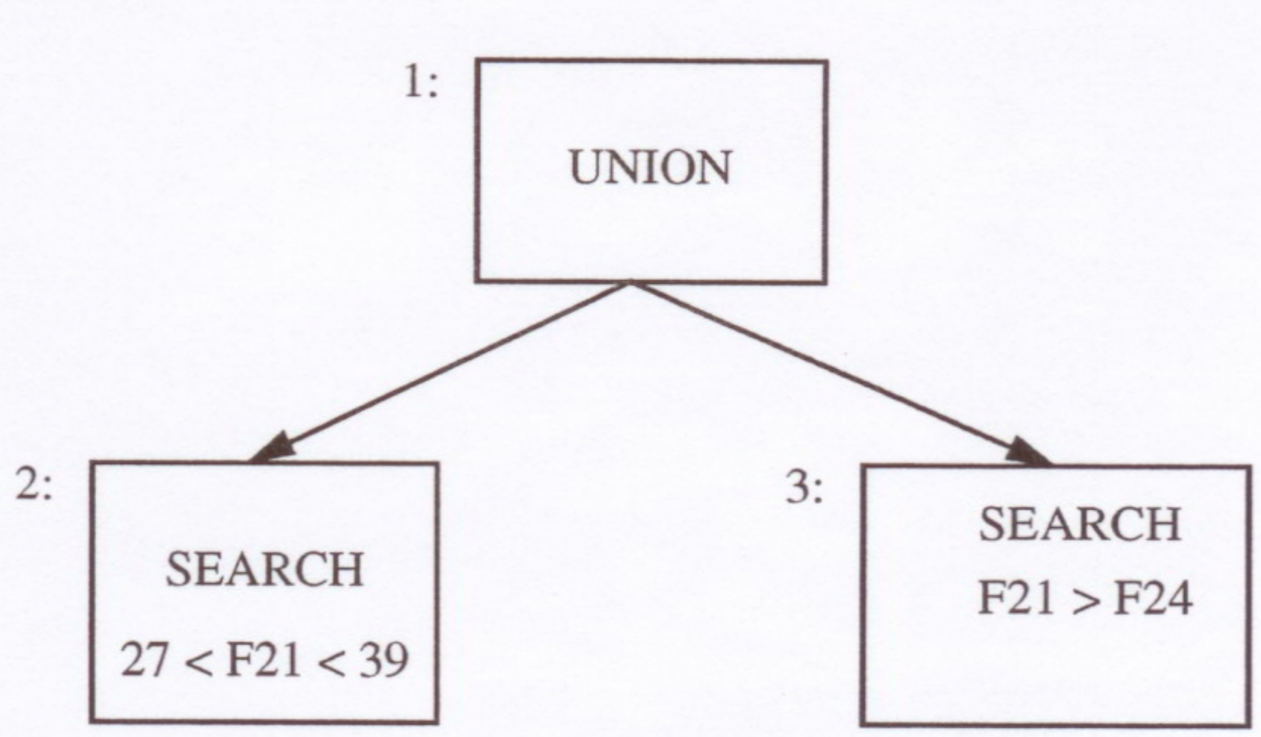
Example query (b).

In English: "Retrieve F22 and F23 from all tuples in R2 having F21 within the range + 27 to +39, or F21 greater than F24".

As an SQL statement:

```
SELECT F22, F23
FROM R2
WHERE F21 > 27 AND F21 < 39 OR F21 > F24;
```

This translates into an IFS/Q type 0 transaction having three nodes:



This query is represented at Interface A as a sequence of 60 32-bit words, whose decimal equivalents are as follows. For readability, the start of the information for each node is asterisked.

60 (transaction length, equal to 60 words following this one)
0 (code for the 'query' type of transaction)
1 * (node_id for the top node)
0 (no other nodes make reference to this node)
2 (identity of the top node's left son)
3 (identity of the top node's right son)

```
12    (the code for the UNION primitive operation)
0     (the UNION's left parameter is not a base table)
0     (the UNION's right parameter is not a base table)
2     (the total number of fields to be returned to the user)
0     (the identity of the first field to be returned)
1     (the identity of the second field to be returned)
2   * (node_id for the left-hand node)
1     (one reference to this node)
0     (this node has no left-hand descendents)
0     (this node has no right-hand descendents)
5     (this is a SEARCH node)
2     (this node searches class_number 2, equivalent to relation R2)
4     (interrogand information for four fields follows)
27    (+27 is the lower-bound for range-matching on the first field)
0     (there is no constant for matching the second field)
0     (ditto, third field)
0     (ditto, fourth field)
10    (the code for 'in the range')
0     wild card for the second field of the interrogand)
0     ditto, third field)
0     (ditto, fourth field)
1     (range information follows for one field only)
39    (+39 is the upper limit for matching the first field)
1     (condition length)
0     (no condition information)
2     (return two fields, identified as follows)
1     (the second field)
2     (the third field)
3   * (identity of the third, or right-hand node)
1     (one reference)
0     (no left-hand descendents)
0     (no right-hand descendents)
5     (this is a SEARCH node)
2     (class_number)
4     (info. for 4 fields follows. NB: no constants are required)
0     (no constant)
```

0 (no constant)
0 (no constant)
0 (no constant)
0 (wild card for this field)
0 (ditto)
0 (ditto)
0 (ditto)
0 (no upper-range required)
7 (length of the condition)
1 (number of components in the disjunction)
1 (number of components in the single conjunction)
4 (the 'greater-than' condition is required)
1 (the first 'arith. expression' is a field)
0 (specifically field F21)
1 (the second expression is a field)
3 (specifically field F24)
2 (return two fields from this SEARCH, identified as follows)
1 (the second field)
2 (the third field).

The above 60-word message is analysed by the Command-Sequence Optimiser - (see Figure 1) - which transforms it into a sequence of five tasks for the IFS/2 hardware. These tasks are transmitted in accordance with Interface B conventions to the Search Module Controller, as follows.

The first task may be represented symbolically as:

2 :: [20 1]

The three 32-bit words transmitted to the SMC are:

2 message length

20 Allocate RAPs
1 one RAP needed

The second task is:

26 :: [5 4 0 0 0 0 0 0 0 0 0 0 7 1 1 4 1 0 1 3 2 1 2 2 4 1 1].

This is transmitted as:

26 message length
5 Search
4 4 field tuple
0 first field is set to zero and
0 other fields set to zero
0 other fields set to zero
0 other fields set to zero
0 first field is wild and
0 other fields are wild
0 other fields are wild
0 other fields are wild
0 no range limits
7 (length of the condition)
1 (number of components in the disjunction)
1 (number of components in the single conjunction)
4 (the 'greater-than' condition is required)
1 (the first 'arith.expression' is a field)
0 (specifically field F21)
1 (the second expression is a field)
3 (specifically field F24)
2 return 2 fields; namely
1 second field
2 and third field
2 class number
4 hash on all 4 fields
1 store results in RAP 1
1 clear RAP 1 before use

The third task is:

21 :: [5 4 27 0 0 0 10 0 0 0 1 39 1 0 2 2 3 2 4 1 0]

21	message length
5	Search
4	4 field tuple
27	first field MAY have constant value 27
0	other fields set to zero
0	other fields set to zero
0	other fields set to zero
10	first field constant value is the lower limit of a range
0	other fields are wild
0	other fields are wild
0	other fields are wild
1	1 upper range limit
39	value is 39
1	condition length
0	no conditions
2	return 2 fields; namely
1	second field
2	and third field
2	class number
4	hash on all 4 fields
1	store results in RAP 1
0	DO NOT clear RAP 1 before use

The fourth task is:

9 :: [12 1 1 4 2 2 1 2 0]

9	message length

```
12      union
1       use the data in RAP 1 as the source
1       store results back in RAP 1
4       number of fields in data
2       number of hash fields (not relevant in this example)
2       return 2 fields
1       second field
2       third field
0       DO NOT clear the RAP before using it
```

The fifth task is:

```
3  ::  [21   4  3]


3       message length
21      show rap
2       no of fields
1       get results from RAP no 1
```

Example query (c)

In English: "Construct a new relation whose fields are (F13, F21, F11, F23), where the relation is made up from only those tuples in R1 and R2 which satisfy the condition that fields (F13 and F11) are equal to fields (F21 and F23)".

The condition is an *equi-join*, expressed symbolically as:

$$R1 \bowtie R2$$

$$3,1 = 1,3$$

As an SQL statement, the whole example (c) is expressed as:

```
SELECT F13, F21, F11, F23
FROM R1, R2
WHERE F13 = F21      AND      F11 = F23;
```

This comes down from interface A as:

37	transaction length of 37
0	transaction type QUERY
1	node id
0	number of references to this node by other nodes
0	left child id
0	right child id
11	operation (JOIN)
1	condition length
0	no conditions
4	return 4 fields, numbered from left to right across all original tuples:
2	the third field
3	the fourth field
0	the first field
5	the sixth field
1	left child class = 1
3	number of fields
0	first field has value 0
0	second field has value 0
0	third field has value 0
0	first field is wild
0	second field is wild
0	third field is wild
0	no range upper limits
1	condition length
0	no conditions
2	right child class = 2
4	number of fields
2	the first field is matched with the third field
0	second field not used
0	third field is matched with first field
0	fourth field ignored
20	perform a join-equals between this field & the third field of the lh child

0 the second field is wild
20 perform a join-equals between this field & the first field of the lh child
0 the fourth field is wild
0 no range upper limits
1 condition length
0 no conditions

When this has been processed by the TOP transputer it will be sent out as a series of five messages. The first message is represented symbolically as:

2 :: [20 2]

This is transmitted as:

2 message length
20 allocate RAP
2 two RAPs needed for this transaction.

The second message is symbolically:

20 :: [5 4 0 0 0 0 0 0 0 0 0 1 0 2 0 2 2 2 1 1]

This is transmitted to the SMC as:

20 message length
5 operation SEARCH
4 number of fields in this relation
0 constant value for first field

0 constant value for second field
0 constant value for third field
0 constant value for fourth field
0 first field is wild
0 second field is wild
0 third field is wild
0 fourth field is wild
0 no ranges
1 condition length
0 no conditions
2 return fields in the following order
0
2
2 class number
2 hash on the first 2 fields before putting into RAP
1 put the results into logical RAP 1
1 clear the RAP before using it

The third message is:

18 :: [5 3 0 0 0 0 0 0 0 0 1 0 2 2 0 1 2 2 1]

18	message length
5	operation SEARCH
3	number of fields in this relation
0	constant value for first field
0	constant value for second field
0	constant value for third field
0	first field is wild
0	second field is wild
0	third field is wild
0	no ranges
1	condition length
0	no conditions
2	return fields in the following order
2	
0	
1	class number
2	hash on the first 2 fields before putting into RAP
2	put the results into logical RAP 2
1	clear the RAP before using it

The fourth message is

16 :: [11 2 1 3 2 2 2 4 1 0 4 0 2 1 3 1]

16	message length

```
11      Join
2       Source of data for one relation is in RAP 2
1       Source of data for other relation is in RAP 1
3       Send the results to RAP 3
2       number of fields in the left hand relation
2       number of fields in the right hand relation
2       number of intersecting fields
4       number of hash fields
1       condition length
0       no conditions
4       return four fields
0
2
1
3
1       clear the RAP (irrelevant in this case)
```

The fifth message is:

```
3  ::  [21   4   3]

3       message length
21      show rap
4       no of fields
3       get results from RAP no 3
```

6. Acknowledgements

It is a pleasure to acknowledge the contribution of Konrad Kok and other members of the IFS group at Essex. The work described in this report has been supported by SERC grant GR/G/54023.

7. References

1. S H Lavington, M E Waite, J Robinson and N E J Dewhurst, "Exploiting Parallelism in Primitive Operations on Bulk Data Types". Proceedings of PARLE-92, the Conference

on Parallel Architectures and Languages Europe, Paris June 1992. Published by Springer-Verlag as LNCS 605, pages 893-908.

2. S H Lavington, J M Emby, A J Marsh, E E James and M J Lear, "A Modularly Extensible Scheme for Exploiting Data Parallelism". Presented at the Third International Conference on Transputer Applications, Glasgow, 1991. Published in Applications of Transputers 3, IOS Press 1991, pages 620-625.

3. S.H. Lavington and J. Wang, "The External Procedural Interface (EPI) for the IFS/2: Parts 1 & 2". University of Essex, Department of Computer Science, Internal Report CSM-164, June 1991.

4. N.E.J Dewhurst and S.H. Lavington, "Relational Algebraic Operations for the IFS/2". University of Essex, Department of Computer Science, Internal Report CSM-168, February 1992.

5. N.E.J. Dewhurst, S.H. Lavington and J. Robinson, "DDB Graph Operations for the IFS/2". University of Essex, Department of Computer Science, Internal Report CSM-169, May 1992.

6. S H Lavington, C J Wang, N Kasabov and S Lin, "Hardware Support for Data Parallelism in Production Systems". Proc. 3rd. International workshop on VLSI for Neural Networks and Artificial Intelligence, Oxford, September 1992, pages 1-12.

7. C J Wang and S H Lavington, 'SIMD Parallelism for Symbol Mapping'. Proceedings of the International Workshop on VLSI for Artificial Intelligence and Neural Networks, Oxford, September 1990, pages C38-C51. Published in: VLSI for Artificial Intelligence and Neural Networks, ed. Delgado-Frias & Moore, Plenum Press, 1991, pages 67-78.

8. S H Lavington, M Standring, Y J Jiang, C J Wang and M E Waite, "Hardware Memory Management for Large Knowledge Bases". Proceedings of PARLE, the Conference on Parallel Architectures and Languages Europe, Eindhoven, June 1987, pages 226 - 241. (Published by Springer-Verlag as Lecture Notes in Computer Science, Nos. 258 & 259).

9. N E J Dewhurst, "IFS/Q Host Library Procedures". Department of Computer Science, University of Essex, May 1993.

APPENDIX A

IFSQ SQL Commands

Key

x y	"x then y"
x|y	"x or y"
[x, y]	"x or y"
[x]	"optional x"
x*	x zero or more times
'..'	literal char
".."	literal string

Grammar

```
COMMAND ::= QUERY
          | "insert into" identifier QUERY
          | "delete from" identifier QUERY
          | "insert into" identifier "values (" FUNC (',' FUNC)* ')'
          | "delete from" identifier "values (" FUNC (',' FUNC)* ')'
          | "create table" identifier '(' CDEC (',' CDEC)* ')'
          | "drop table" identifier

QUERY   ::= "select" SLIST "from" FLIST ["where" WCOND]

SLIST   ::= '*' | SITEM (',' SITEM)*
SITEM   ::= identifier ".*" | "count(*)" | AGGFN '(' FUNC ')' | FUNC
```

```
AGGFN   ::= "count" | "max" | "min" | "sum" | "avg"

FLIST   ::= FITEM (',' FITEM)*
FITEM   ::= identifier [identifier]

WCOND   ::= PRED | '(' WCOND ')' | "not" WCOND | WCOND ["and", "or"] WCOND
PRED    ::= FUNC COMP FUNC | FUNC ["not"] "in (" QUERY ')' | "exists (" QUERY ')'
COMP    ::= '=' | "!=" | '>' | ">=" | '<' | "<="

FUNC    ::= number | string | identifier ['.' identifier] | '(' FUNC ')' |
            FUNC ['+', '-', '*', '/', '%'] FUNC | '-' FUNC
CDEC    ::= identifier ": int"
          | identifier ": string"
```

Notes

1. 'number' is a string of digits.
2. 'string' is anything enclosed in double quotes.
3. 'identifier' begins with '_' or a letter, & may also contain digits
4. If several commands are present in one input stream, they must be separated by semicolons, so:
   % ifsq
   <command> ; <command> ; .. < command> ; <command>
   ^D
5. What is missing from IFS SQL:
   (a) ORDER_BY, GROUP_BY & HAVING clauses;
   (b) comparison with (subquery), SOME(subquery) or ALL(subquery).
   (c) Some of the specialised SQL data formats such as *money* and *date*.

APPENDIX B

IFS/Q Host Library Procedures

N.E.J. Dewhurst
Department of Computer Science, University of Essex

Issue 1: 12th May 1993

This technical note describes the applications-programmer interface implemented by the IFS/Q library. The IFS/Q is a version of the IFS/2 knowledge-based server that has been designed to give direct support to SQL. The IFS/Q is a hardware unit which is attached via a SCSI channel to a standard host computer such as a Sun workstation. General descriptions of the IFS/2 architecture will be found in references 1 and 2. A detailed description of the lower-level software interfaces for the IFS/Q will be found in reference 3.

The IFS/Q library consists at present of 22 functions, written in C, for manipulating the following:

- the IFS/2 hardware unit;
- schema files & data files (on host disk);
- the *current schema*, a data structure maintained by the library routines that describes the current IFS contents;
- a software Lexical Token Converter (LTC).

The IFS/Q library provides a general platform upon which SQL-oriented information systems may be built. The present library supports applications in which end-users manipulate their database via standard SQL statements ~ i.e., SQL statements represented as ASCII strings. In the future, it is intended to offer programmers a lower-level alternative entry point, intended for systems in which it is more convenient to express queries using C structures representing relational algebraic expressions.

An example of an application of the IFS/Q library is shown diagrammatically in Figure 1. This is the IFS/2's SQL language interface, as provided for demonstration and performance-measurement purposes. Figure 1 illustrates the flow of information between an end-user and the IFS/2 add-on unit. The 'interface A' of Figure 1 is formally defined in reference 3.

The rest of this Technical Note is organised as follows. First we show how certain IFS/Q library functions relate to concepts such as database schema; communication with the IFS; relational tables; the LTC. Then we give a complete list of all 22 C library functions.

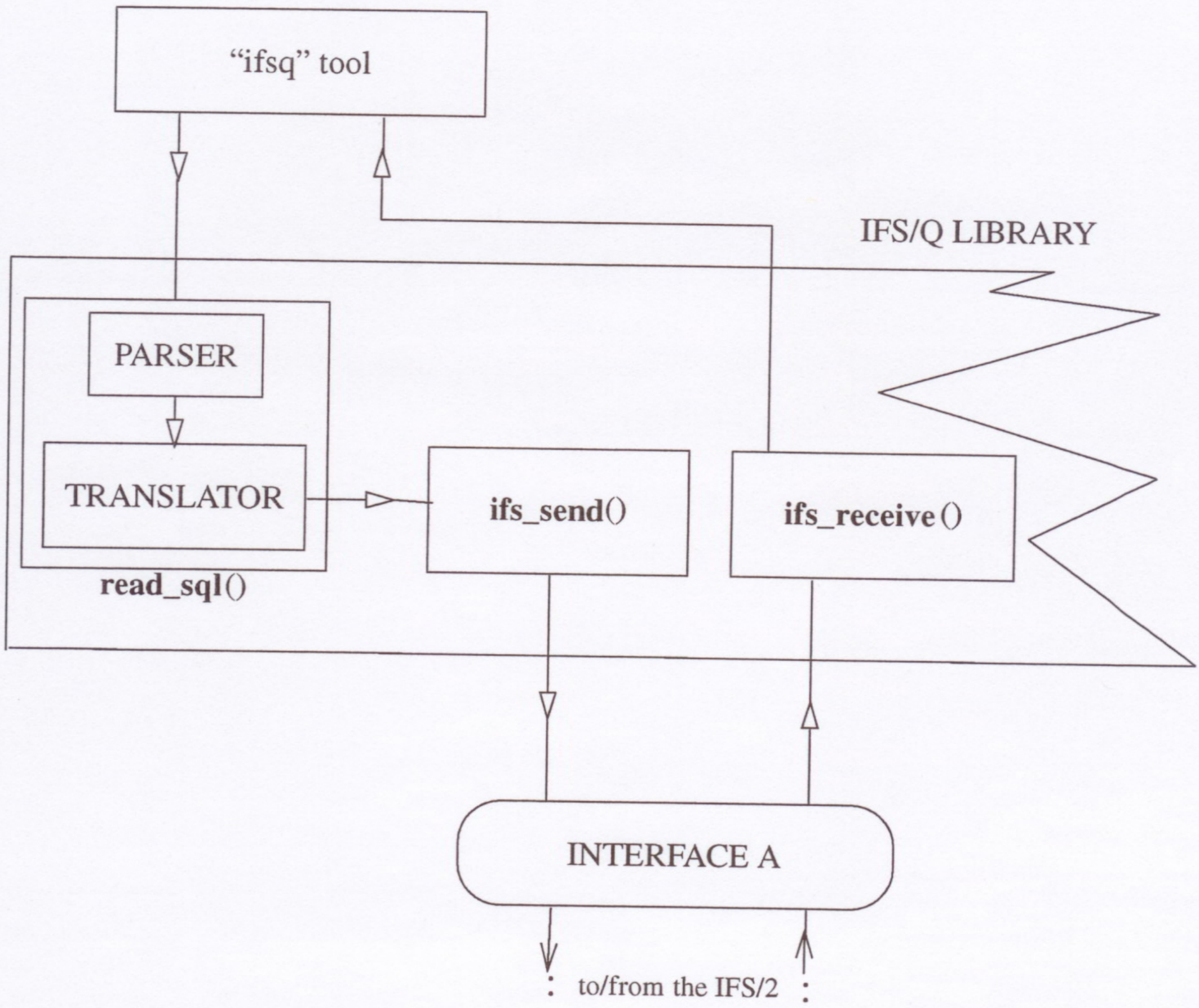


Figure 1: an example of an application running on a host computer

Current Schema

This is consulted by the **read_sql**() function when it translates table and column names into IFS class numbers and column numbers. So tables referenced in SQL queries must be in the current schema.

Tables get into the current schema in two ways:

(i) **ifs_create_table**() and **ifs_drop_table**(), which respectively declare and undeclare a single table. The appropriate transaction is sent to the IFS,

(ii) **ifs_load_schema**(), which replaces the current schema by a schema read from a named schema file.

Note on creating & dropping tables: one could send a CREATE_TABLE transaction to the IFS (see ref. 3) by compiling a create-table command using **read_sql**(), then sending it using **ifs_send**(). But the current schema would not be updated, so the new table would be invisible to subsequent SQL commands. Similar comments apply to DROP-TABLE.

The IFS

ifs_create_table() and **ifs_drop_table**() both communicate with the IFS, to allow it to keep up its internal list of active class numbers.

read_sql() reads one SQL query from the stdin stream, compiles it into a form suitable for sending to the IFS and returns a pointer to the buffer holding the compiled code. At present the IFS supports most of the common SQL constructs - (see also ref. 3). We have not yet implemented the following:

(a) ORDER_BY, GROUP_BY, & HAVING clauses;

(b) comparison with (subquery), SOME (subquery), or ALL (subquery);

(c) some of the specialised SQL data formats such as *money* and *date*.

ifs_send() sends compiled code to the IFS.

ifs_receive() reads a response from the IFS. It separates responder fields from diagnostic data, and places both in a Response struct.

The interface also provides a macro DELAY(x), which waits for x seconds (where x may be, and typically is, a fraction). This can be used to give the IFS time to respond before calling **ifs_receive**().

See also notes on **ifs_load_table**() below.

Schema Files

Each line in a schema file is parsed as:

<white space> <first component> <white space> <second> <white space> <rubbish>.

"Rubbish" means that everything following the third lot of white space is ignored, so this is one way to include comments. A '#' in its first column causes a line to be ignored: this is the other way to include comments.

Each entry should include the following, *in this order:*

table <name of table>

in_ifs <class number>

<name of first field> <type of first field>

<name of 2nd field> <type of 2nd field>

...

<name of last field> <type of last field>

The 'table' line indicates the start of an entry: the entry extends to the next 'table' line, or to EOF.

<type> may be 'int' or 'string'.

If you want a component to include white space, you can enclose it in either single or double quotes.

Several functions are provided for manipulating schema files directly (adding and deleting entries, etc.).

Data Files

ifs_table_load() reads tuples field-by-field from a named file and writes them to a given class in the IFS. The fields should be in ASCII format, separated by white space.

Note: an IFS class has to be declared using a CREATE_TABLE transaction before data can be loaded into it - (see ref. 3).

Software LTC

This is a separate module providing five functions: **ltc_void**(); **ltc_load**(); **ltc_save**(); **ltc_str2id**() to convert strings to tokens; and **ltc_id2str**() to convert tokens to strings.

Summary of IFSQ Functions

Note on Types

Structs Table and Response are defined in header "ifsq.h", as is enumerated type ReturnCode. All IFS/Q functions return a ReturnCode: the possible values are error; no_error; found; and not_found.

When an error is detected, the globally-available variable last_ifs_error is set to the corresponding error code.

Functions

ifs_ping(void)

ifs_reboot(void)

Not yet implemented.

ifs_use_phys(void)

ifs_use_sim(void)

Toggle between use of the physical IFS (i.e. hardware) and the software simulator.

ifs_create_table(Table *t)

All of t's fields other than in_ifs are filled in. Copies the class number provided by the IFS into in_ifs, and the whole entry into the current schema.

Returns error, no_error.

ifs_drop_table(char *t_name)

Deletes entry for table called t_name from current schema, and releases the corresponding IFS class number.

Returns error, no_error.

ifs_send(int *buf)

Write code to the IFS. buf[0] contains the length of the following code.

Returns error, no_error;

ifs_receive(Response *r)

Reads a response from the IFS; fills in r appropriately.

Returns error, no_error.

ifs_load_schema(char *file_name)

ifs_save_schema(char *file_name)

Return error, no_error.

ifs_load_table(int c_no, int nf, char *file_name)

c_no is the target class number, nf the number of fields per tuple. The file contains a stream of fields; it is sent to the IFS as a stream of *tuples*, hence the need for nf.

Returns error, no_error.

ifs_save_table()
Not yet implemented.

read_sql(int **buf)
The address of the compiled code is placed in the object pointed at by buf.
Returns error, no_error.

ltc_void(void)
Returns no_error.
ltc_save(char *file_name)
ltc_load(char *file_name)
Return error, no_error.
ltc_str2id(char *str, unsigned *id)
Returns error; found if converted from existing entry; not_found if new token issued.
ltc_id2str(unsigned id, char *str)
Returns error; found if converted OK; not_found.

schema_add(char *db_name, Table *t)
Makes a new entry in file db_name by copying data out of t.
Returns error, no_error.

schema_remove(char *db_name, char *t_name)
Deletes an entry from file db_name. t_name names the table.
Returns error; no_error if deleted OK; or not_found if entry wasn't present.

schema_lookup(char *db_name, Table *t)
t has its name field filled in. Locates the corresponding entry in db_name and copies the rest of the entry into t.
Returns error, no_error or not_found.

schema_check(char *db_name, char *t_name)
Looks in db_name for entry for table named by t_name.
Returns error, found or not_found.

References

1 **S H Lavington, M E Waite, J Robinson and N E J Dewhurst,** "Exploiting Parallelism in Primitive Operations on Bulk data Types". *Proceedings of PARLE-92, the conference on Parallel Architectures and Languages Europe, Paris, June 1992. Published by Springer-Verlag as LNCS 605, pages 893-908.*

2 **S H Lavington,** "A Novel Architecture for Handling Persistent Objects". *Microprocessors and Microsystems, Vol. 17, No. 3, April 1993, Pages 131 -138.*

3 **S H Lavington, N E J Dewhurst, A J Marsh, J M Emby and D R Thoen,** "An SQL interface for the IFS/2 knowledge-base server: release 1." *University of Essex, Department of computer Science, Internal Report (CSM-180), December 1992.*