

Design Description

Software: FORTRAN support in T4 code generator
Software Version: T4/T8 native F77 V1.1
Design Version: 3
File: ALEF::D1:[IMS.DOCS.DESIGN]FTNICODE.LAY

[Rev 002 17-JUN-87, ADC] added cacheing of COMMON block pointers for F77 V1.1
[Rev 001 6-NOV-86, ADC] updated after informal design review (ADC/PSR/IAY/NPH, 31-OCT-86)
Changed GMOVE to use perm.
[Rev 000 22-OCT-86, ADC] created

why are § 4.2, 4.3
about P3, P2 resp?
just wondering... -lan
Looks OK to me.

Copyright © 3L Ltd, 1987-1992

1 Objectives and Scope

The software design proposed in this document will allow the transputer (T4) back end to be extended to handle the FORTRAN-specific additions to I-code defined in the documents listed below.

2 Related Specifications and Documents

Fortran Items in I-Code, Peter Robertson, 5 August 86
SYSDOC_ICODE_Working.Notes, Revision 1.4, 7 July 86

3 Product

This design will be implemented in the beta release version of the FORTRAN compiler for Inmos.

4 Design Description

The I-codes to be added or modified for FORTRAN are

DEFINE	declare general area
AREA-SIZE	declare size of general area
DEFINE-AT	declare object within general area
POSITION	initialise general area
INIT	support for POSITION
FIXUP	pointer initialisation in general areas
PROMOTE	I-code stack manipulation
GMOVE	string move with blank-padding
CONVERT	explicit type conversion

The first group deals with the new concept of a "general area". The second group are miscellaneous support operations.

General areas are named blocks of static read/write storage which allow the first pass of the FORTRAN compiler to control storage allocation for variables. The first pass can place a variable at a given byte offset from any general area using the new DEFINE-AT I-code. Note that the first pass is therefore responsible for observing any alignment restrictions imposed by the target machine: in particular, for the T4 transputer, offsets for non-byte variables must be word aligned (multiples of four).

Variables in general areas have no defined values at the start of execution, unless they are explicitly initialised using the POSITION and INIT I-codes described later.

General areas can be *internal* or *external*.

Internal areas are local to a module. The name of an internal area is irrelevant and is not used. Different internal areas might be defined to store scalar variables, arrays, and constants, for example.

External areas are global to all modules of a program; they correspond to COMMON blocks in FORTRAN. An external area declaration can be either a *definition* or a *reference*. References to external areas are generated using general area DEFINE items with the spec bit set. Definitions do not have the spec bit set. All references to an external area, AREA1 say, in any program module, refer to the same block of static storage called AREA1. This block of storage is allocated automatically if no external area definition of AREA1 is found in any module. Referenced areas cannot be

initialised by INIT. To initialise an external general area (as in a BLOCK DATA subprogram) the area must be declared as an external area definition rather than an external area reference. It is an error for a program to contain more than one definition for the same external area.

The size of all general areas must be defined using the AREA-SIZE I-code. Internal areas, and variables allocated in them using DEFINE-AT, may be used before the size of the area has been defined. In particular, definition of area sizes may be left until the end of the I-code stream.

4.1 Run-Time Storage of General Areas

Internal areas and external area definitions will be treated as an extension of the existing static storage area (GLA). For external area definitions, a linker DATASYM directive will be output indicating the displacement of the area from the start of the static area as a whole.

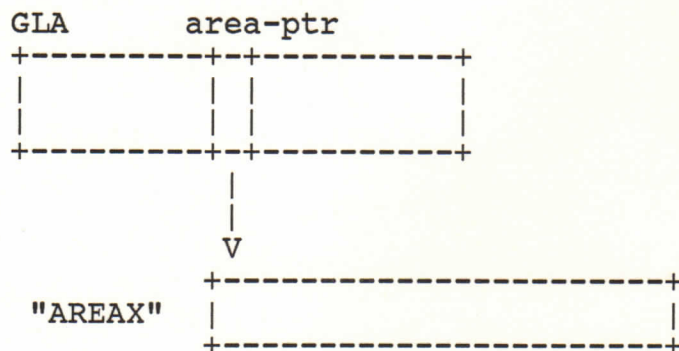
```
+-----+-----+-----+-----+
| static (own) variables | area #1 | area #2 |...| area #N |
+-----+-----+-----+-----+
```

The module STATIC area size declared to the linker in the output object file and to the run time STARTUP module in the module's Initialisation Control Block will include the sizes of any internal areas or external area definitions, and the GLAP area will contain initialisation information for the whole of the static storage area, including any internal areas.

Variables allocated storage by DEFINE-AT will be accessed like normal static variables, via word 0 of the display vector in the local stack frame. For example, to load a variable I previously declared by DEFINE-AT to be allocated at a displacement of 40 words from the start of an internal area which is itself at a displacement of 120 words from the base of the static storage area as a whole. (The total offset is known at compile time, see below).

```
ldl    display[0]      ;pointer to base of GLA
ldnl   40+120          ;load I
```

Variables allocated in referenced external areas are accessed via a pointer in GLA to the base of the external area. Linker directives are output to cause the GLAP word corresponding to the pointer to be patched with the displacement of the referenced area, which is declared to the linker as a COMMON block (its size derived from the AREA-SIZE I-code). Fixup information must be added to the run time relocation area to cause the displacement information in the pointer word to be converted to an absolute pointer during run time startup.



Loading a variable I at a displacement of 20 words from the start of AREAX becomes:

```
ldl    display[0]    ;pointer to base of GLA
ldnl   area-ptr      ;pointer to external area
ldnl   20             ;displacement from area base
```

This was the scheme used up to V1.0.1 of F77. In V1.1, one indirection is eliminated by caching pointers to external areas in the stack frame, so the code is simplified to:

```
ldl    cached-ptr    ;pointer to external area
ldnl   20             ;displacement from area base
```

On subroutine entry, the *area-ptr* for each external area used by that subroutine is copied into a new area in the stack frame: the CACHE AREA. The code to do this is generated by pass three (see the description of the BLK CACHE directive in section 4.2 below) and is output as part of the main entry sequence for the subroutine, and for each ENTRY statement it contains.

4.2 Pass Three

Pass three of the compiler must be extended to handle general areas.

Pass two allocates an *area-number* to all general areas which are not declared with external spec attributes (i.e. all areas for which space must be allocated in the static data area of the module being compiled: internal areas and external area definitions).

Pass three already recognises some fixed codes for areas in the object file and the stack frame; these area codes are used in the pass three directives which generate code to load and store variables (DIR LOAD etc). To allow these existing directives to handle access to variables in general areas, the *area-number* codes for general areas must be distinguishable from the existing codes: negative values will be used for general *area-number* codes.

These area numbers are used by pass three as indexes into a new GENERAL AREA table. Table size will not be a fixed limit as pass three will be told the size of the table (via the GLOBAL record), and can allocate a dynamic array of the appropriate size.

During the initial PROCESS BLOCKS phase of pass three, each table entry will hold the size in bytes of the corresponding area. In order for these sizes to be available at this point, pass two must write area size information into the "blocks" stream read by pass three (see BLK AREA SIZE below). At the end of this phase the sizes will be converted into displacements from OWN AREA BASE (by a new procedure, ALLOCATE GENERAL AREAS). The table can then be used during the subsequent PROCESS DIRECTIVES phase to convert <area-number,offset> pairs generated by pass two into displacements from the start of the static data area (i.e. ordinary static variable references). The VAR ACCESS procedure in pass three must be extended to handle (negative) general area numbers by using the table.

The new directives which will be added to pass three are listed below.

BLK AREA SIZE *area-number num-bytes*

This is a new directive for the "blocks" stream read by pass three. It defines the size of an area for which storage must be allocated in this module.

p3 builds list
of cache'd words
for each
block.

BLK CACHE *cache-disp static-disp*

This blocks stream directive causes pass three to generate code in the entry sequence of the current procedure to copy the word at *static-disp* bytes from the start of the static area to the word at *cache-disp* bytes from the start of the cache area in the procedure's stack frame. The CACHE AREA is a new area in the stack frame, which will be allocated immediately before the local scalar area (as these cached pointers to external COMMON blocks are expected to be heavily used by FORTRAN programs).

DIR X AREA DEF *area-number string*

This directive causes pass three to output a DATASYMB item in the object file defining the symbolic name *string* as the location in the static data area of the base of the general area given by *area-number*.

NB: using BLK CACHE is an implicit use of the static area: P2 needs to do a 'USE(0)' after issuing it.

DIR COMMON *num-bytes xno*

This directive causes pass three to output a COMMON item to the object file indicating that this module accesses a COMMON block of size *num-bytes*. The name of the block (required in the corresponding linker COMMON item) is given to pass three as an *xno*: an index into the EXTERNAL table. The EXTERNAL table entry given by *xno* must have been previously initialised by a DIR X REF directive from pass two. Using *xno* in this way avoids the necessity for pass two to remember the name of an external area between its declaration and the AREA-SIZE I-code which tells it how large the block will be. Instead, when the block is declared, pass two issues a DIR X REF with the name of the block, and remembers only the corresponding *xno*.

DIR POSITION *area-number byte-offset*

The existing pass three's DIR SELECT AREA directive does not allow positioning within an area: after selecting an area, data in DIR CODE directives is simply appended to any existing data in that area.

This new directive allows pass two to set a current position at a given offset from any general area for which space has been allocated in this module. Subsequent DIR CODE directives will initialise the area starting from the given offset.

DIR AREA FIXUP *area-number offset flag*

This directive is used to implement the new FIXUP I-code. It causes pass three to dump a word into the currently selected area which is the actual displacement of *area-number* from OWN AREA BASE, plus the *offset* (in bytes). The *flag* controls whether the total displacement dumped is in words or bytes. It is an error if *flag* indicates that the displacement should be given in words and the *offset* is not word aligned.

The normal usage of this directive will be to dump the actual displacement from OWN AREA BASE of variables subject to FIXUP directives into the relocation table processed during run time module initialisation. The directive is required because pass two, which builds the relocation table, only knows the addresses of variables to be fixed up as <area,offset> pairs and cannot calculate the offset from OWN AREA BASE of a general area.

Note the SELECT AREA routine in pass three must be extended to handle the new dummy "ordinary" area required by INIT (see 4.3.5 below). It does not need to output a linker ADDRESS directive when this area is selected: the dummy select can simply be ignored.

4.3 Pass Two

This section describes the processing pass two must do for each new I-code.

4.3.1 DEFINE (T=15)

General areas given the attributes external spec by DEFINE (T=15,X=3,S=0) will have the following information stored in the descriptor.

TYPE	GENERALS (to allow some usage checking)
FLAGS	X DATA SPEC flag set
DISP	displacement from start of GLA of the pointer to the external area
EXTRA	<i>xno</i> of area identifier name of the area
CACHE DISP	displacement from start of cache area of the pointer to the external area
BLOCK NUM	block sequence number of block in which CACHE DISP is valid

CACHE DISP and BLOCK NUM were added in V1.1 to support cacheing of external area (COMMON block) pointers in the local stack frame. BLOCK NUM identifies the block which most recently used the general area associated with the descriptor; CACHE DISP is the offset of the cached pointer in the cache area of that block's stack frame. The BLOCK NUM field allows multiple uses of the same general area within one block to use the same cached pointer. The first time a particular external spec'd general area is used in a new block its BLOCK NUM is different from the current block sequence number. In that case, pass two allocates the next word in the current block's cache area, overwrites CACHE DISP with its offset, outputs a BLK CACHE pass three directive to initialize it with the word at offset DISP in the static area, and sets BLOCK NUM to the current block sequence number. The current block sequence number is maintained in a global variable BLOCK SEQUENCE NUM, which is incremented by ASSEMBLE on entry to a new procedure. The block sequence numbers start from 1 and the BLOCK NUM field is initialized to 0, so that the first time an area is used the BLOCK NUM is certain to differ from the current block sequence number, causing a cache word to be allocated and initialized.

CACHE DISP overlays the FORMAT descriptor field; BLOCK NUM overlays AREA.

Because the linker COMMON directive must contain the string identifier of the area, the string must be saved until the corresponding AREA-SIZE I-code is processed. This is done by allocating an external number, *xno*, and passing the string through to pass three (DIR X REF) to store in the EXTERNAL table entry given by *xno*. The string can thereafter be referred to (in a DIR COMMON directive) via the *xno*. The EXTRA field of the descriptor is used to hold the *xno*.

Descriptors for internal areas declared by DEFINE (T=15,X#3) and external area definitions declared by DEFINE (T=15,X=3,S=0) contain the following information.

TYPE	GENERALS
FLAGS	X DATA SPEC not set
AREA	pass 3 area-number (negative)

Area-numbers are allocated by DEFINE VAR when new internal areas (or external area definitions) are declared.

For external area definitions (T=15,X=3,S=0), pass two must output a DIR X AREA DEF directive using the current external identifier as the string.

For external area references (T=15,X=3,S=1), the next word in GLA is allocated, and a DIR X DSPEC directive is output by DUMP EXTERNAL to patch it with the displacement of the area given by the current external identifier string. An entry must

be made in the run time relocation table area indicating that this GLA word is to be relocated.

4.3.2 AREA-SIZE tag size

The *tag* is a VAR table index; the indicated descriptor must have been declared as a general area (check that TYPE=GENERALS).

Note: multiple definitions are not checked, in order to avoid problems if a module contains multiple COMMON statements which refer to the same block.

Size is the size in bytes of the area.

If the X DATA SPEC bit of the descriptor's FLAGS word is set, a DIR COMMON directive is output to pass three (the EXTRA field contains an *xno* referring to the name of the area). Otherwise, a BLK AREA SIZE directive must be written to the "blocks" stream to make pass three allocate space for the area in this module.

4.3.3 DEFINE-AT

This new code will be handled by calling DEFINE VAR which after handling the type and form of the variable being defined (DEFV) will look to see if the I-code currently being processed is DEFINE-AT rather than DEFINE. If it is, the other fields in DEJV will be set up like this:

AREA	<d> area tag from DEFINE-AT
DISP	<e> displacement from DEFINE-AT
FLAGS	new BASED VAR FLAG set

The new BASED VAR FLAG indicates to STACK VAR that it must stack a descriptor derived from both the VAR being stacked and the general area descriptor whose VAR table index is given by the AREA field of the VAR being stacked.

For example,

```
DEFINE 17, [general-area]
DEFINE-AT J 99, [int] [area=17] [disp=40]
STACK 99
```

When the STACK operation occurs, the VAR table will look like this:

VAR	DISP	AREA	FLAGS
#17		-3	0
:	:	:	:
#99:	40	17	BASED VAR

STACK VAR(99) must stack a VinS descriptor with DISP=40, AREA=-3, and the other descriptor fields set up as for an ordinary static variable.

The situation is more complex if the X DATA SPEC flag is set in the area descriptor. This is when the CACHE DISP and BLOCK NUM fields in the area descriptor are used.

VAR	DISP	AREA (BLOCK#)	CACHE DISP	FLAGS	
#17	8	13	32	XDATASPEC	<--+
:	:	:	:	:	
#99:	40	17		BASED VAR	-----+

Prior to V1.1, STACK VAR would stack a VinRec descriptor: 40(8(static-area)). Here 8(GLA) is the location of the word pointing to the external area.

In V1.1, STACK VAR uses the cached pointer: 40(32(cache-area)). We assume the current BLOCK SEQUENCE NUMBER is 13, i.e. area 17 has already been used in this block, and the CACHE DISP of 32 is valid. If we were in a different block, CACHE DISP would have to be set up first as described in section 4.3.1 above.

Note: arrays are assumed to be handled by pass one, using explicit pointer operations, and so DEFINE-AT cannot be used to declare structured objects (arrays, records, or nested general areas).

4.3.4 POSITION *area offset type*

This I-code item sets the OWN TYPE flag used by the INIT and ADUMP routines to the pass two internal type code derived from the *type* argument, checks that the VAR entry given by *area* does not have the X DATA SPEC flag set, converts the *area* to a pass three area-number (the area-number is stored in the VAR entry), and saves the area-number and offset as the new global INIT-POSITION (see INIT below) in a pair of global variables.

Note that declaration of an ordinary own object resets INIT-POSITION to the start of that object. This must be handled by DEFINE.

[Note that the type codes used with POSITION and CONVERT are different from the type codes which must be used with the INIT-TYPE I-code]

4.3.5 INIT

If INIT-POSITION has not been set by a previous POSITION I-code, the behaviour of INIT is unchanged (data is dumped into the constant area or the own area depending on the setting of the OSTATE flag).

If INIT-POSITION has been set by a previous POSITION I-code, a DIR POSITION directive must be issued to pass three to ensure that data dumped by ADUMP will be stored in the correct place.

Before issuing the DIR POSITION directive, any pending output to the currently selected area must be flushed, and the current area CA protected from disturbance in ADUMP, by selecting a dummy area. The CA variable for this area will be the global variable used to hold the offset part of the current INIT-POSITION, so that ADUMP will automatically update INIT-POSITION as required.

Note that initialisation "to the address of TOS if TOS is a variable" as defined in SYSDOC_ICODE_Working.Notes V1.4 is not allowed (in any implementation).

4.3.6 FIXUP *tag*

The *tag* must be a general area. It may not be a "static object" as specified in SYSDOC_ICODE_Working.Notes V1.4. Also, the current INIT-POSITION must be in a general area. FIXUP cannot be applied to ordinary own variables.

Assuming that the *tag* is restricted to being a general area, there are two cases. Both require that the word identified by the current INIT-POSITION be relocated before run time by adding in the base address of the general area given by *tag*.

1. *Tag* is an external area reference.

Pass two must add an entry to the relocation table area indicating that during run time startup the GLA word holding the pointer to the external area is to be added in to the word indicated by INIT-POSITION. Since pass two cannot calculate an actual GLA displacement from the <area-number,offset> pair in INIT-POSITION, a pass three DIR AREA FIXUP directive must be used to put the GLA displacement of the word to be relocated into the relocation table area.

2. *Tag* is an internal area or an external area definition.

In this case, pass two must create a relocation table entry specifying the GLA displacements of both the word to be relocated and the area given by *tag*. DIR AREA FIXUP must be used to generate both displacements.

4.3.7 CONVERT

The type codes are converted to pass two internal types as follows.

<n>	Type	Format
1	INTEGERS	4
2	BYTES	1
3	INTEGERS	4
4	REALS	4
5	LREALS	8
6	BYTES	1
7	BYTES	1

An internal temporary variable of the appropriate type and format is allocated, and CONVERT TYPE is used to convert TOS to this type and assign the result to the temporary. The temporary replaces TOS.

4.3.8 GMOVE

If the source-length is greater than or equal to the destination-length, MOVE BYTES will be used to move destination-length bytes from source to destination.

Otherwise, a perm routine will be called to move MIN(source-length,destination-length) bytes from source to destination and fill any remaining bytes in the destination string with ASCII spaces.

4.3.9 PROMOTE

Consists simply of pointer manipulations on the I-code stack.

4.3.10 Procedure Parameters

FORTTRAN procedure parameters cannot be represented using the IMP/Pascal procedure parameter mechanism. The mechanism employed to support pointers to functions in C can be used instead.

This represents a pointer to a function (e.g. a procedural argument) as a simple integer value (which is a pointer).

To generate a pointer to a procedure, push the procedure descriptor on the I-code stack and apply the ADDRESS function. The resulting pointer (a full length integer) is left on the top of the stack. This pointer can then be passed as a value argument to a FORTRAN subroutine in place of the procedural argument.

To call the procedure pointed to by such an argument, use the FMAP I-code. This converts the integer descriptor on the top of the stack into a function descriptor for a function with C argument-passing conventions. This should suffice for FORTRAN as all arguments are just integer addresses. Arguments are passed to this function using ASSIGN-PARAMETER as usual. The function must be called using the VARIABLE-CALL instruction (because it is a C function with a variable-length argument list). If necessary, the function result must be popped from the I-code stack using the POP instruction.

Example:

```
SUBROUTINE X(R)
CALL R(I)
```

This would become:

```
Define X [void routine]
Start
  Define R [integer simple]
Finish
Define-At I ...
Stack R
Fmap 1    {convert to integer fn}
Stack I
Address
Assign-Parameter
Variable-Call
Pop      {unwanted integer result}
```

5 Assumptions and Limitations

The I-code support described in this document is only intended for use with the EPC FORTRAN front end; it will not be used by other Lattice compilers.

It is assumed that FORTRAN arrays are handled by the front-end using explicit pointer manipulation. Thus, array declarations are not supported by DEFINE-AT.

COMMON blocks can be initialised by defining a DATASYMB for a block of ordinary static storage which the linker must ensure that all COMMON items with that name refer to. If COMMON items with no corresponding DATASYMB definition exist, the

linker must allocate the storage for them. We must be able to reference this storage using linker REF and LONG DATAFIX items in the usual way (in order to initialise pointers to external general areas).

For example, consider two modules A and B, which both refer to a common block FRED:

```
A:  COMMON "FRED" 100
    REF "FRED"
    LONG 20 DATAFIX 1  -- 20(GLA) is pointer to FRED

B:  COMMON "FRED" 50
    REF "FRED"
    LONG 8 DATAFIX 1   -- 8(GLA) is pointer to FRED
```

The linker must allocate enough space (100 words) for FRED, and modules A and B must be able to get pointers to that space.

Note that a single module may contain more than one COMMON item with the same name, for example:

```
SUBROUTINE A
COMMON /BLOCK/ A, B, C
END

SUBROUTINE B
COMMON /BLOCK/ D, E, F, G
END
```

If these two subroutines were compiled from a single source file, the output object file would contain two COMMON items giving the size of BLOCK as 3 or 4 words, as well as two REF items (and corresponding DATAFIX patches) referring to the symbol BLOCK.

In addition, the linker must allow for a BLOCK DATA subprogram, C, being used to create and initialise the space for FRED:

```
C:  DATASYMB "FRED" 10  -- 10(GLA) is start of FRED
```

This definition of FRED must cancel the effect of the COMMON items in modules A and B.

Note that there is no way in this scheme of checking that the programmer has allocated enough space in the BLOCK DATA subprogram for the largest COMMON item which will refer to the space from any of the other modules in the program.