SOFTWARE SUPPORT FOR
MOTOROLA 68000 MICROPROCESSOR AT CERN

CERN CONVENTION FOR
PROGRAMMING THE MC68000 FAMILY

Working Party
reported by B. Carpenter, edited by R. Cailliau

GENEVA
1984
Propriété littéraire et scientifique réservée pour tous les pays du monde. Ce document ne peut être reproduit ou traduit en tout ou en partie sans l'autorisation écrite du Directeur général du CERN, titulaire du droit d'auteur. Dans les cas appropriés, et s'il s'agit d'utiliser le document à des fins non commerciales, cette autorisation sera volontiers accordée.

Le CERN ne revendique pas la propriété des inventions brevetables et dessins ou modèles susceptibles de dépôt qui pourraient être décrits dans le présent document; ceux-ci peuvent être librement utilisés par les institute de recherche, les industriels et autres intéressés. Cependant, le CERN se réserve le droit de s'opposer à toute revendication qu'un usager pourrait faire de la propriété scientifique ou industrielle de toute invention et tout dessin ou modèle décrits dans le présent document.

© Copyright CERN, Genève, 1984

Literary and scientific copyrights reserved in all countries of the world. This report, or any part of it, may not be reprinted or translated without written permission of the copyright holder, the Director-General of CERN. However, permission will be freely granted for appropriate non-commercial use.

If any patentable invention or registrable design is described in the report, CERN makes no claim to property rights in it but offers it for the free use of research institutions, manufacturers and others. CERN, however, may oppose any attempt by a user to claim any proprietary or patent rights in such inventions or designs as may be described in the present document.
Abstract

The CERN convention for programming the MC68000 family of microprocessors gives a set of rules describing the layout of the memory and stack frames used by routines as they should appear before and after their calling sequences. It does not deal with the instructions used to achieve these states.

The aim of the convention is to allow programming language mixing as well as debugging of programs built from units written in different languages. It is to be followed by programmers and programming-language compilers.
CONTENTS

Purpose and History 1

1. GENERAL CONSIDERATIONS 2

2. BACKGROUND INFORMATION ON THE MC68000 3
   2.1 Data types in memory 3
   2.2 System stack 4
   2.3 Some relevant instructions 4
      2.3.1 BSR - Branch to subroutine 4
      2.3.2 JSR - Jump to subroutine 5
      2.3.3 RTS - Return to subroutine 5
      2.3.4 LINK - Link and allocate 5
      2.3.5 UNLK - Unlink 6
   2.4 Memory management considerations 6
   2.5 Peculiarities of new members of the MC68000 family 6

3. DATA TYPES 7
   3.1 Boolean 7
   3.2 Byte (character) 7
   3.3 Word (short integer) 7
   3.4 Long Word (long integer) 7
   3.5 Real 8
   3.6 Strings 8
   3.7 Arrays 8

4. WORKING AREA LAYOUT 9
   4.1 Layout of the global area 10

5. ROUTINE CALLS AND STACK FRAME LAYOUT 11
   5.1 Passing of parameters 11
   5.2 The standard stack frame layout 12
   5.3 Actions involved in a routine call 15
   5.4 Format of code 15
6. EXCEPTION HANDLING 16
7. BIBLIOGRAPHY 16
8. GLOSSARY 17
INDEX 20
Purpose

This document describes a programming convention for the MC68000 microprocessor family. The aims of the convention are to allow communication between routines written in different languages (language mixing), and to allow re-use of software (including in particular the MoniCa debugger). In particular, library routines must follow the convention precisely.

History

This document was drafted by K.Osen, B.Carpenter, and R.Cailliau (PS) on the basis of documents produced by J.Blake and H.von Eicken (DD) during 1982. The final version was edited following the meetings held during 1983-84 by a working group chaired by R.Cailliau. Besides the persons mentioned above, other active members of the group were:

- P. Anderssen (SPS)
- R. Keyser (SPS)
- G. Shering (SPS)
- L. Tremblet (DD)
- P. van der Stok (SPS)
- D. O. Williams (DD)

and useful comments were received from R.D.Russell (University of New Hampshire), H. von der Schmitt, and others.
1. GENERAL CONSIDERATIONS

This is a description of the CERN convention for programming the MC68000 family of microprocessors. It describes the "core" convention which must be adopted by all language implementations and by assembly language programmers.

Disclaimer

The convention is intended to allow language mixing, and such mixing is guaranteed to work only within the range of facilities covered by the convention. This document does not attempt to cover all cases of useful programming. In particular, it does not suggest how the operating system should handle the hardware or indeed even whether an operating system should be used.

The convention does not cover all cases of language mixing, especially since certain data types may have been defined in incompatible ways by language designers (see for example the Boolean type).

However, routines whose interfaces adhere to the convention should have a higher rate of acceptance by users. Programs using such routines should be easier to develop and debug.

Each language will probably require its own extensions to the convention, but these should be kept to a minimum. They should take the form of explanations on how data types and routine types particular to the language are represented in the 68000. It is clear that interfacing to such particularities from a different language is a tricky business and the result will probably not be portable (see also introduction to chapter 3).

The manual for each language is the obvious place where these rules are put.
2. BACKGROUND INFORMATION ON THE MC68000

[A glossary at the end of this document explains relevant terms and abbreviations.]

2.1 DATA TYPES IN MEMORY

The MC68000 uses three operand sizes in memory accesses: Byte, Word and Long Word. These are 8, 16 and 32 bits respectively.

A memory access is controlled by the memory address and the operand size. In the case of words and long words the memory addresses always have even values (except for the MC68020). The transfer is done as shown in the following examples:

```
```

A statement such as "1(A5):=MUB of D3" means: the middle upper byte (bits 16 to 23) of D3 is moved to the address contained in A5 plus one. See Glossary for other abbreviations. The following diagram illustrates the MC68000 addressing convention:

```
| 31 24 23 16 15  8  7  0 | bit numbers
| HOB | MUB | MLB | LOB |
| 0   | 1   | 2   | 3   |
```

Address of long word

- 3 -
2.2 SYSTEM STACK

The stack in the MC68000 grows downwards, that is, from high addresses towards low addresses. This convention is built into the instruction set of the MC68000.

The MC68000 system (main) stack pointer (SP) is address register A7. During routine calls (JSR or BSR instructions) the return address is stored using the "Address Register Indirect With Predecrement" addressing mode on SP. This works as follows:

Before the call SP points to the last entry on the stack. The term "points" means that a long word (in memory or a register) holds the address of the most significant byte of an item. The first action in the call is to decrement SP by 4 since it takes 4 bytes to store the return address. The return address is thereafter stored at the address now contained in SP.

Upon return from the subroutine (RTS instruction), the return address is fetched using the "Address Register Indirect With Postincrement" addressing mode. After having used SP to pick up the return address, SP is incremented by 4. After this final action the stack is in just the same state as right before the call.

SP may only contain even addresses; otherwise an address error exception will occur.

2.3 SOME RELEVANT INSTRUCTIONS

2.3.1 BSR - Branch to Subroutine

The address of the instruction immediately following the BSR instruction is pushed onto the system stack. This address is the value of PC during the execution of the BSR instruction. Thereafter a two's complement integer is added to PC. Depending on the required length of the jump, an 8 or 16 bit displacement integer must be chosen.

In the case of the 8 bit displacement the length of the BSR instruction is two bytes, and the jump range relative to the address of the BSR instruction is -126 to +128. Displacement 0 is invalid since it in fact signals a 16 bit displacement.

In the case of the 16 bit displacement the length of the BSR instruction is four bytes, and the jump range relative to the address of the BSR instruction is -32766 to +32768.
If additional range is required, the JSR instruction should be used instead. In practice, the range restriction on BSR means that it should be used only to call routines inside the same code module; JSR should be used for external routines.

2.3.2 JSR - Jump to Subroutine

JSR works similarly to BSR, except that JSR may use all the MC68000 memory addressing modes. JSR allows subroutine calls to the whole 24 or 32 bit address space. This is particularly convenient for calls to shared library routines.

Example: JSR FRED

2.3.3 RTS - Return from Subroutine

RTS moves the long word indexed by SP into PC, and increments thereafter SP by four (i.e. pops the return address from the stack). Execution then resumes at the new address in PC.

Example: RTS

2.3.4 LINK - Link and Allocate

The LINK instruction reflects the development of high level, stack oriented languages. The instruction works as follows:

The instruction assumes the usage of stack frames. A stack frame is a piece of memory allocated to every invocation of a routine. The stack frame associated with the currently executing routine is called the current stack frame. The local variables of a routine are stored within its stack frame. An address register must be reserved to point to the current stack frame. This address register is called the current stack frame pointer. In the CERN convention, A6 is reserved for this purpose.

The LINK instruction is executed as one of the very first instructions of a routine. The value of A6 (which initially points to the previous stack frame), is pushed onto the system stack (using the predecrement mode on SP), and the contents of SP is thereafter copied into A6. A two's complement 16 bit integer is finally added to SP in order to allocate local memory to the just-called routine. The value of this integer must
be fixed at compilation or assembly time; it will of course be negative due to the direction of growth of the stack.

The long word now pointed to by A6 is called the dynamic link of the current stack frame. All stack frames have a dynamic link. Together they form a continuous chain beginning with A6.

Example: LINK A6,#-40

2.3.5 UNLK - Unlink

UNLK is the inverse instruction of LINK. The UNLK instruction is used just before the RTS (Return from Subroutine) instruction in a routine. UNLK works as follows:

The value of A6 is copied into SP. This effectively releases the local space earlier allocated to the active routine. Thereafter a long word is popped off the stack and copied into A6. This restores the old value of A6.

Example: UNLK A6

2.4 MEMORY MANAGEMENT CONSIDERATIONS

Memory management is related to the convention in only one way. It is necessary that a run-time system and debugger must be able to access program code as read-only data; thus use of the MC68000 function codes for addresses in program code space must be limited to write-protection, since a restriction to execute-only mode would prevent effective debugging.

2.5 PECULIARITIES OF NEW MEMBERS OF THE MC68000 FAMILY

It is known that long word operations in the MC68020 will be significantly more efficient if aligned on long word boundaries. Exploiting this will require modifications in the code generators, but will not affect the convention.

The MC68010 and the MC68008 have no known implications for code generators or the convention.
3. DATA TYPES

GENERAL WARNING: sharing sophisticated data types between languages is rather difficult due to the special characteristics of each language. Whenever possible, library routines should stick to simple types: bytes, integers, reals, and one-dimensional arrays of known length.

The FORTRAN-77 standard requires that INTEGER, REAL and LOGICAL each occupy one 'numerical storage unit'; they therefore have the same default size. This size is 4 bytes in the CERN convention. This means, in particular, that LOGICALs must be exchanged with other languages as if they were long integers. FORTRAN-77 also requires that characters each occupy one 'character storage unit'; this is one byte in the CERN convention.

3.1 BOOLEAN

The Boolean type occupies one byte.

The variable is FALSE if all bits are zero. It is recommended that TRUE be represented by the integer value 1 (LSB=1), but a Boolean with any non-zero value must always be interpreted as TRUE. Rationale: Pascal and Modula-2 require the use of these ordinal values.

3.2 BYTE (CHARACTER)

The byte type takes 8 bits of storage. A frequent use of byte data is the storage of 7 bit ASCII characters, right-justified without parity.

3.3 WORD (SHORT INTEGER)

The word type takes 2 bytes of storage and must always be stored on an even memory address. When used as a signed integer, twos complement format applies.

3.4 LONG WORD (LONG INTEGER)

The long word type takes 4 bytes of storage and must always be stored on an even memory address. When used as a signed integer, twos complement format applies.
3.5 REAL

Several different real number formats are possible:

- IEEE 32 bit format.
- Motorola 32 bit FFP (Fast Floating Point).
- Norsk Data ND-100 48 bit format (accelerator divisions only).
- IEEE 64 bit format.

The IEEE formats should be used whenever possible. The documentation of assembly language routines should state which form of real number is expected.

3.6 STRINGS

A string is a linear array of bytes with variable length, but correct transfer of string parameters is not guaranteed by the convention. Certain languages use string descriptors.

3.7 ARRAYS

Arrays of all types are stored linearly, consecutively, and contiguously. The address of an array is considered to be the address of its first (lowest subscripts) existing element, regardless of the subscript values.

In some languages [including Pascal] the last written subscript varies fastest ("row-major order"); in others [including FORTRAN] the first written subscript varies fastest ("column-major order"). Because of this, assembly-coded and library routines should use one-dimensional arrays whenever possible.

There is no standard descriptor, and the CERN convention does not cover conformant (compliant) arrays.
4. WORKING AREA LAYOUT

The following diagram gives the layout of the working area for a given task, except for data areas which are accessed via their absolute addresses (shared data or common blocks). The CERN convention does not consider shared data (thus library routines should not rely on shared data). Every task possesses a working area.

High address

A5 ===========>
| Global area |
| initial or |
| global stack |
| frame |

A6 ===========>
| Current |
| stack frame |

A7 ===========>
| Free space |

stack limit ==> 256 bytes margin

heap top ------> allocated heap space
heap origin pointer ==> (end of working area)
Low addresses
4.1 LAYOUT OF THE GLOBAL AREA

The global area is a permanent attribute of a task. It contains important variables used by the run time systems of different languages, but not normally accessible from user programs. The CERN convention requires that A5 points to byte $000$ of the global area, which is laid out as follows:

<table>
<thead>
<tr>
<th>High addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>$080\text{(A5)}$ register save area etc.</td>
</tr>
<tr>
<td>$07F\text{(A5)}$ 31rst static vector</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>$00C\text{(A5)}$ 3rd static vector</td>
</tr>
<tr>
<td>$008\text{(A5)}$ Heap free list pointer</td>
</tr>
<tr>
<td>$004\text{(A5)}$ Heap origin pointer</td>
</tr>
<tr>
<td>$000\text{(A5)}$ Stack limit pointer</td>
</tr>
</tbody>
</table>

Or, somewhat more specifically:

- $080+$ register save area if required for context switching
- $00C-07F$ (29 x 4 bytes) static level pointers ("display")
- $008$ (4 bytes) heap free list pointer (if needed)
- $004$ (4 bytes) heap origin pointer
  (lowest even address in data area)
- $000$ (4 bytes) stack limit pointer
  (even address of last free byte for stack)

NOTE: a safety margin of 256 bytes must be maintained between the stack limit and the highest used heap address. This margin is intended to allow run-time support routines to use a few bytes of stack without formality.

If running in supervisor state, it will also be used by exception vector service routines (which must therefore have modest stack use to avoid
overflow), and A7 must then be kept in the lowest position in order to avoid overwriting of data when a trap occurs. (Note: in the CERN recommended operating system RMS68K, these remarks apply to user state, since user-provided interrupt service routines run using the stack of whichever user task is active when the interrupt occurs).

5. ROUTINE CALLS AND STACK FRAME LAYOUT

5.1 PASSING OF PARAMETERS

Parameter descriptions are stacked from last to first immediately before calling a routine. This means that the last parameter description has a higher address than the first parameter description.

The last-to-first approach of stacking parameters is important because it allows the called routine to locate the first parameter. This is essential for C in which the number of parameters is allowed to vary. There is no mandatory indication of the number of actual parameters, but it is recommended that this indication be passed in D0 if required by an implementation. Because of possible variations in the length of parameter lists, the called routine cannot clean up the stack before returning and it is the caller's responsibility to do so.

The following diagram shows the state just before a routine call (JSR or BSR):

```
High address:

    Old stack
    __Actual_param_n__  }
    _______  }  Actual parameter
    __Actual_param_2__  }  area
    __Actual_param_1__  }

    A7 ==========>

    Free space

Low address:
```

A call-by-reference parameter description requires 4 bytes, while a call-by-value parameter description normally has a size equal to that of the actual parameter. In the case of byte data passed by value, a full word is allocated on the stack since "MOVE.B source,-(A7)" actually word-aligns the stack pointer.

The value of a function is returned in D0 if it occupies no more than 4 bytes. If the returned value needs more space than 32 bits, an implicit first call-by-reference parameter is introduced.
The fifth item is the address of the exception handler of the routine concerned (see below). This long word must be present, and must be set to zero if there is no exception handler.

The local working space is allocated by the routine which owns the stack frame. It provides all necessary storage for the routine; its length depends on the code of the called routine. Its layout and contents are language-dependent.

- Its first byte is at offset -$00D from A6;
- Its first short word is at offset -$00E from A6;
- Its first long word is at offset -$010 from A6.

The actual parameter area is used only during calls to other routines, and its size depends on the actual parameters to be passed.
5.3 ACTIONS INVOLVED IN A ROUTINE CALL

When a routine is to be called, the calling routine must first expand its own stack frame with an actual parameter area by stacking the parameter descriptions as described above. The actual parameter area must be removed by the calling routine after the called routine has returned.

The called routine is expected to preserve the registers A5, A6 and A7. This is a natural consequence of their use in the convention. No other registers need to be saved.

The actions involved in a routine call can be summarised by the following pseudo code:

Calling routine:

    push__parameters__on__stack;
    call__routine(routine__name);
    pop__parameters__off__stack;

Called routine:

    build__stack__frame;
    routine__body;
    remove__stack__frame;
    return;

5.4 FORMAT OF CODE

The code of a routine must satisfy the following layout:

```
DC.W  language code  * see below
DC.W  'name of routine' * pad to even length with trailing blank
DC.B  length of name  * zero or even
DC.B  static level   * $FF if none; <3 to stop tracebacks
Entry_Pt ...  * first executable instruction
```

Further information for debugging will be put in a separate place and used by a debugging system. The workings of any debugger are not part of this convention.
HOB: High order byte. HOB consists of bits 24 to 31 of any long word.

LOB: Low order byte. LOB consists of bits 0 to 7 of any byte, word or long word.

LSB: Least significant bit. LSB is always bit 0.

Local variables: These variables are allocated inside a stack frame in the working area.

MC68000: This is a 16 bit microprocessor standardised at CERN. The MPU has 16 multipurpose registers each 32 bits wide. The address bus has 24 bits, allowing the access of 16777216 bytes.

MC68008: Software-compatible with the MC68000, but with an 8-bit external data path.

MC68010: This is a 16 bit microprocessor which is pin and software compatible with MC68000. A few additional privileged instructions are included in order to manipulate the memory management system.

MC68020: This is a 32 true bit microprocessor which will be software compatible with MC68000. It has a 32 bit data path and a 32 bit addressing space.

MLB: Middle lower byte. MLB consists of bits 8 to 15 of any word or long word.

MPU: Micro processing unit. The MC68000 is an MPU.

MPU processing states:
The MC68000 MPU has two processing states: the Supervisor State and the User State.

MSB: Most significant bit. The bit number of MSB depends on the operand size.

MUB: Middle upper byte. MUB consists of bits 16 to 23 of any long word.

PC: Program counter. This special register has 32 bits. During the execution of an instruction, PC points to the next instruction.

Procedure: A routine without a function value.

Process: Code which can be executed by a task.

Rn: Register number n. This notation is used when any address or data register can be used.

Routine: A routine is a common term for procedure and function.
Segment: A segment is a contiguous piece of memory.

SP: System stack pointer. This another name for address register A7, both in user and supervisor state.

SSP: Supervisor stack pointer. This is another name for A7 while the computer is in the supervisor state.

Static level: In a nested block-structured language such as Pascal, the level of nesting of a routine in the source file. For practical reasons, the outermost level is 2, so a doubly-nested routine is at level 4. Static levels 3 to 31 are valid (numbering is from 1, 2 is not necessary).

Static vector: Pointer to the currently relevant stack frame for a certain static level. 29 such pointers are stored in the global area; when a routine on static level N is entered, it saves the previous static vector N and replaces it with A6. Variables which are neither global nor local are accessed via the static vectors.

Subroutine: A subroutine is a routine in MC68000 assembly terminology.

Supervisor state: The supervisor state is a privileged state. This state is used by the operating system, and allows use of the privileged instructions. All exception handling takes place in the supervisor state.

Task: Execution instance of a process, possesses a working area, is created by the operating system.

TRUE: This Boolean state is represented by a byte containing any non-zero value; preferred value is +1.

User state: Normal programs execute in the user state of the MPU. The user state forbids use of the privileged instructions.

USP: User stack pointer. This is another name for A7 while the computer is in the user state.

Working area: Contiguous piece of memory allocated to a task when this is created, and where the task finds its data. This area must be both readable and writeable from the state (supervisor or user) in which the task runs.
<table>
<thead>
<tr>
<th>Index Entry</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrays</td>
<td>8</td>
</tr>
<tr>
<td>Boolean</td>
<td>7</td>
</tr>
<tr>
<td>BSR</td>
<td>4</td>
</tr>
<tr>
<td>byte</td>
<td>7</td>
</tr>
<tr>
<td>call by reference</td>
<td>11</td>
</tr>
<tr>
<td>call by value</td>
<td>11</td>
</tr>
<tr>
<td>code format</td>
<td>15</td>
</tr>
<tr>
<td>data types</td>
<td>3, 7</td>
</tr>
<tr>
<td>debugging</td>
<td>15</td>
</tr>
<tr>
<td>dynamic link</td>
<td>6</td>
</tr>
<tr>
<td>error code</td>
<td>16</td>
</tr>
<tr>
<td>error types</td>
<td>16</td>
</tr>
<tr>
<td>exception handler</td>
<td>13, 16</td>
</tr>
<tr>
<td>exception handling</td>
<td>16</td>
</tr>
<tr>
<td>FORTRAN-77</td>
<td>7</td>
</tr>
<tr>
<td>function value</td>
<td>11</td>
</tr>
<tr>
<td>global area</td>
<td>10</td>
</tr>
<tr>
<td>JSR</td>
<td>5</td>
</tr>
<tr>
<td>language code</td>
<td>15</td>
</tr>
<tr>
<td>language mixing</td>
<td>2</td>
</tr>
<tr>
<td>LINK</td>
<td>5</td>
</tr>
<tr>
<td>long word</td>
<td>7</td>
</tr>
<tr>
<td>memory management</td>
<td>6</td>
</tr>
<tr>
<td>operating system</td>
<td>2</td>
</tr>
<tr>
<td>parameters actual</td>
<td>13</td>
</tr>
<tr>
<td>parameters formal</td>
<td>13</td>
</tr>
<tr>
<td>parameters number of</td>
<td>11</td>
</tr>
<tr>
<td>parameters passing</td>
<td>11</td>
</tr>
<tr>
<td>real</td>
<td>8</td>
</tr>
<tr>
<td>routine calls</td>
<td>11, 15</td>
</tr>
<tr>
<td>RTS</td>
<td>5</td>
</tr>
<tr>
<td>stack frame</td>
<td>5, 12</td>
</tr>
<tr>
<td>stack frame global</td>
<td>12</td>
</tr>
<tr>
<td>stack frame initial</td>
<td>12</td>
</tr>
<tr>
<td>stack pointer</td>
<td>4</td>
</tr>
<tr>
<td>static vector</td>
<td>13</td>
</tr>
<tr>
<td>string</td>
<td>8</td>
</tr>
<tr>
<td>system stack</td>
<td>4</td>
</tr>
<tr>
<td>UNLK</td>
<td>6</td>
</tr>
<tr>
<td>word</td>
<td>7</td>
</tr>
<tr>
<td>working area</td>
<td>9</td>
</tr>
</tbody>
</table>