The MTS Coding Conventions

by

Steve Burling

The University of Michigan
Computing Center
1075 Beal Avenue
Ann Arbor, Michigan 48109-2112

January 6, 1984

Table of Contents

I. Introduction

This document describes the MTS coding conventions. These conventions were originally invented at UBC and published under the name of "Resource Manager Coding Conventions." Despite the specific nature of the name, the conventions actually described a coding system that was generally applicable to assembler programming. Subsequent proliferation of non-Resource-Manager code built according to these conventions has made them a de facto standard in some quarters of the MTS community. Accordingly, the name of the conventions was changed to delete the specific Resource Manager reference.

The major goal of these conventions is to support a coding environment that encourages consistent style and clean hierarchical code structure. This is accomplished mainly by implementing selected concepts of modern high-level languages using a set of coding rules and a set of assembler macros. Unlike some other macro-supported coding techniques, these conventions concentrate on the larger code objects, such as global storage areas and procedures. Below the procedure level, local coding style is mostly left up to the programmer. Thus, these conventions are compatible with most sets of so-called "structured programming macros."

This document describes the current version of the coding conventions. Differences between this version and earlier versions can be found in Section VI. The MTS job program now uses the current version, and work is underway at UM to convert existing uses of the "old" coding conventions to use the current coding conventions. The relatively small amount of additional support required for use inside the MTS assemblies is included in the MTS-specific macro library. The specific additions to the coding conventions for use inside MTS are described in Section VII.

The ultimate goal of all this is to end up, once again, with only one version of the coding conventions.

NOTE: Although most of this document is couched in terms of assembly language, a distinction must be made between the coding conventions themselves and the assembly language macros that provide an implementation of the coding conventions. The PLUS and Pascal/JB programming languages generate code that conforms to the coding conventions as described in this document. The GOM programming language generates code that conforms to a previous version of the coding conventions. The user of any of these languages need not know the finer details of the coding conventions, except when interfacing to programs written in languages that do not support them, such as FORTRAN.

II. Structure

The following linkage conventions and storage rules provide a block-structured environment with simple variable scope rules and a linkage/local-store stack mechanism. Here are the concepts:

A. ASSEMBLY

An assembly consists (mainly) of one or more **routines.**

B. ROUTINE

A routine is a procedure. Every routine must be declared (by macro -- see below for its description) before it is called or used. Every routine definition (i.e. code) is **closed** -it has a definite beginning and a definite end, and all of the routine's executable code lies between them. Furthermore, no routine definition resides inside another routine definition, except for subroutines which are described below. There are two kinds of routines, **routines** and **subroutines.**

1. Routines

A routine is self-addressable, callable by BALR, movable between assemblies, and externally callable if declared to be.

2. Subroutines

A subroutine is a routine that is attached to a particular routine. It is coded physically inside its parent, but outside the definition of any other subroutine. This is the exception to the nested-definition rule that we mentioned above. It is self-addressable. It saves registers, but uses the same local stack frame as its parent. The registers are saved in what would be the stack frame provided to a called routine, and the next stack frame pointer is updated appropriately.

C. STORAGE MANAGEMENT

The coding conventions attempt to define a strict repertoire of storage types. These types are described next.

1. Global Storage

Global storage is storage that is global to more than one routine. It is pointed to by R11, which in general is left untouched during subroutine calls. It is possible, however, for a routine declaration to request a different global storage pointer to be loaded into R11 before it is called (the original R11 value is restored from the stack after the called routine returns).

The global storage pointed to by R11 must always be at least 8 bytes long, since the first two words of this area are, by convention, assumed to contain a pointer to the CLS transfer vector and a pointer to the global stack descriptor, respectively.

The management of routine global storage is left to the particular group of routines which use it. Many programs use the pseudo-register concept, which, although not ideal, has a few advantages over the more usual method of using a global DSECT. All pseudo-register references are made via macros. Although UMLOAD supports pseudo-register-vector lengths up to 16 megabytes, for efficiency the macros assume the length is a page or less. This limitation does not exist for programs written in PLUS. It is not expected that this limitation will ever be a serious one, since working storage is available elsewhere -- see #2 below -- and since large global data structures should only be rooted in the storage addressed by R11, not allocated in it.

For applications where multiple program loads must refer to the same pseudo-register-vector definition, the macros that declare pseudo-registers can be made to produce an UMLOAD LCSPR table that communicates the format of one load's pseudo-register vector to a subsequent load. This situation will exist most commonly when a non-resident program calls a resident routine.

2. Local Storage

Storage local to routines is allocated on stacks. Stack storage is roughly equivalent to PL/I AUTOMATIC storage, but without the INITIAL option. (The effect of the INITIAL option is available by using the DCI macros.) Local storage exists only across one invocation of each routine. Allocation and de-allocation of storage of this same scope is the responsibility of the routine. Subroutines use their parent routines' stack frames.

The term "RSECT" is used to refer to the DSECT that describes a routine's stack frame.

Routine local storage that must exist across calls will usually be set up as a control block or blocks that are directly or indirectly pointed to by a parameter to the routine being called. Of course, for routines that have only one invocation per context, permanent local storage can be chained from global storage.

3. Other storage

The coding conventions have no rules about storage other than global storage and local storage. Presumably, routines will get and free control blocks and other dynamically-allocated pieces of storage as they need to .

D. NAMING CONVENTIONS

Each routine's name, the labels within it, and the variables in its RSECT should begin with a globally unique 3 character prefix. The default names for the RSECT, the save area within it, and the register area within the save area are constructed from the first 3 characters of the routine name, unless the SYMBOL_PREFIX option of the ROUTINE macro is used, q.v. This means, for example, that a routine named TEST will have, by default, an RSECT named TESRCT, which will contain a save area named TESSA. TESSA will have as fields within it TESR0...TESR15 (in that order).

E. TRANSFER VECTOR

Frequently, certain routines need services whose implementation varies depending on who the caller is. The way the coding conventions support this context-dependent variation is by having a transfer vector. In each context, initialization code establishes a transfer vector that contains pointers to the appropriate service routines for that context. This transfer vector contains addresses for only those service routines which vary from one context to another, not for every routine or even every service routine. The transfer vector is part of the global storage, as described above. Each transfer vector entry is a separate pseudo-register, and contains the address of the routine to call, and the address that the called routine expects to be in R11 at the time it is called.

III. Register Usage

Here is a table of how registers are used in the coding conventions:

Register	At call	In routine	At return
R0	R-parameter	Scratch	Return value (or restored)
R1	R-par or A(S-pars)	Scratch	Return value (or restored)
R2	R-par	Scratch	Return value (or restored)
R3	R-par	Scratch	Return value (or restored)
R4-R9	Undefined	Available	Restored
R10	Undefined	Local base	Restored
R11	A(global storage)	A(global storage)	Restored
R12	A(Caller's stack frame)	A(Current stack frame)	Restored
R13	A(Callee's stack frame)	A(Next stack frame)	Restored
R14	A(Return to caller)	Available	Restored
R15	A(Callee)	Available	Return code (or restored)

In the "In Routine" column of this table, certain registers are labeled as "Scratch" and others as "Available." The spiritual difference between scratch registers and available registers is that scratch registers should be used only for very local purposes whereas available registers should cater to more lofty and stable applications.

Return values in Registers 0-3 are optional. If a routine has fewer than 4 return values, the remainder of R0-R3 at return are restored to their contents at the time of the call.

The parameters and return values in R-type calls must each be dense register sets, i.e. register N shall not be a parameter register unless register N-1 is one, with a similar rule applying to return values.

IV. Linkage

A. ENTRIES AND EXITS

Both routines and subroutines are self-addressable, register-saving, BALR-called routines. Each subroutine is considered to exist in the code space of a containing routine. No more than one routine can call a given subroutine.

Routines are of two types, INTERNAL and EXTERNAL. Both of these types, and subroutines, are called in the same way, namely:

```
...       ...                    (R13 = A(next stack
                                 frame))
L         R15,A(the routine)
BALR      R14,R15
```

ROUTINE entry and exit sequences are

Entry:

```
USING     myrsect,R13
STM       R0,R15,mysa            (save in my dsect)
LR        R10,R15
USING     me,R10                 (local base)
LR        R12,R13                (A(current stack frame))
DROP      R13
USING     myrsect,R12
LA        R13,myrsectlength(,R12)(A(next stack frame))
```

Exit:

```
L         R0,returnvalue         (perhaps)
.
.
L         R3,returnvalue         (perhaps)
L         R15,returncode         (perhaps)
LM        Rn,R14,mysa+Rn*4       (or R15)
BR        R14
```

where Rn is the first register that doesn't contain a return value.

SUBROUTINE entry and exit sequences are

Entry:

```
          STM      R0,R15,0(R13)        (save in next stack
                                        frame)
          LR       R10,R15
          USING    me,R10               (local base)
          LA       R13,4*16(,R12)       A(next stack frame)
```

Exit:

```
          L        R0,returnvalue       (perhaps)
          .
          .
          L        R3,returnvalue       (perhaps)
          LA       R14,4*16             (back up stack)
          SR       R13,R14
          L        R15,returncode       (perhaps)
          LM       Rn,R14,mysa+Rn*4     (or R15)
          BR       R14
```

where Rn is the first register that doesn't contain a return value.

Transfer-vector routines are called as follows:

```
          ...      ...                  (R13 = A(next stack
                                        frame))
          RPR      L,R15,PRNAME.        (routine address)
          RPR      L,R11,PRNAME+4.      (its global storage)
          BALR     R14,R15
          L        R11,mysa+R11*4       (restore our R11)
```

B. PARAMETERS

There are two types of routine parameters. One is standard OS variable-length S-form, the other is standard R-form with at most four parameters in registers 0-3.

C. RESULTS

There are two types of routine results. One is the register result. A routine with an R-type calling sequence can have up

to four results in registers 0-3. A routine with an S-type calling sequence can have zero results, or one in register zero. The other type of result is the normal S-type result, with a value passed back through one of the parameters.

V. Assembler Implementation

The assembler implementation of the coding conventions consists of an extensive set of macros. Consistent with our use of some high-level-language techniques in object program organization, some high-level-language ideas have been borrowed for source program structure. Specifically, all routines must be declared prior to their definition or invocation. Routines are declared using a special macro that registers attribute information in global symbols and symbol arrays. Routine calls and returns are all done by macros that do attribute-checking according to the declarations of the routines they refer to. Routines must be strictly delimited -- there are delimiting macros that do structure-checking with the help of global symbols. Similarly, RSECTS and pseudo-register vectors are defined and delimited by structure-checking macros.

A. SHORT LIST OF MACROS

Here is a summary list of the macros that implement the coding conventions:

ROUTINE	To declare a routine
SUBROUTINE	To declare a subroutine
RBEGIN	To begin a routine
REND	To end a routine, dsect, or pseudo-register vector
RENTER	To begin a routine and enter it
RCALL	To call a routine
RRETURN	To return from a routine
REXIT	To return from a routine and end it
RSPARSET	To set up the parameters for an S-type call
RSPARLOC	To define the storage for an S-type parameter list
RSECT	To begin an RSECT
RSA	To define a routine’s save area
RDSECT	To define a DSECT
REGS	To generate register equates
RRCTST	To test a return code
RPRV	To define a pseudo-register vector
RPRDEF	To define a pseudo-register
RTVENTRY	To define an entry in a transfer vector
RPR	To generate an RX or RS instruction that references a pseudo-register

RPRI	To generate an SI instruction that references a pseudo-register
RPRC	To generate an SS instruction that references a pseudo-register or registers
RNEXT_STACK	To switch to the next stack in the "stack of stacks"
RPREV_STACK	To switch to the previous stack in the "stack of stacks"
RPUSH	To push assembler options onto an assembly-time stack
RPOP	To pop assembler options off of an assembly-time stack
RPUSHSTACK	To grab stack at the end of the current stack frame
RPOPSTACK	To pop the stack back to the location saved by a previous RPUSHSTACK
RSET	To set options for use by other macros
RDISPLAY	To display the attributes of a routine.
CCPUNT	To punt after an error

B. RULES FOR USING THE MACROS

Here are the general rules for use of the macros:

1. A routine must be declared before it is called and/or defined.
2. A routine is delimited by its initial RBEGIN or RENTER macro and its terminal REND macro. Any RCALL or RRETURN macro that appears outside a routine is a structure error. Note that REND terminates a routine but RRETURN doesn't.
3. A ROUTINE can't be defined inside another ROUTINE. A SUBROUTINE must be defined inside the ROUTINE to which it belongs. However, no SUBROUTINE can be defined inside another SUBROUTINE. A RBEGIN or RENTER macro that appears in such a position as to violate this rule is a structure error.
4. An RSECT must begin with an RSECT macro and end with a REND macro.
5. A pseudo-register definition must begin with an RPRV macro and end with a REND macro.
6. Except for SUBROUTINES, no routine, RSECT, or transfer vector can be defined inside another routine, RSECT, or transfer vector. (Actually, an RSECT can be defined inside the routine that it belongs to. See the RSECT macro description.)

C. OS LINKAGE

In addition to generating calls, exits, and returns for coding conventions routines, the macros include fairly complete support for OS linkages. An external routine declared to be of linkage-type "OS" (see the ROUTINE macro, below) can be called by the normal RCALL macro, and an internal routine declared to be of linkage-type "OS" will be callable by a standard OS program.

The implementation of this mechanism takes two forms depending on whether the coding conventions routine is calling the OS routine or is called by the OS routine. Parameter types in these conventions are, of course, compatible with those of OS programs, so parameter interfacing is never required.

When a coding conventions routine is calling an OS routine, the OS routine must be provided with an OS save area pointed to by R13. In the coding conventions, R13 points to the next stack frame which is suitable for an OS save area. It is also necessary that the stack descriptor be updated before calling an OS routine, so that if the OS routine calls back into the coding conventions environment, the next available location on the stack may be used.

When a coding conventions routine is being called by an OS routine, a stack must be allocated. When an internal routine is declared to be of linkage-type OS, the RENTER macro generates code to find the stack and global storage. How this is done is determined by the PSECT option of the ROUTINE macro, q.v.

D. DETAILED DESCRIPTIONS OF MACROS

The following pages give specific documentation on each of the macros. Many of these macros accept parameters in either a functional or a keyword format. The descriptions describe the keyword names for the keyword format. In the functional format, underscores in the keywords are replaced with hyphens. In new applications, the keyword format should be used, since it assembles much faster. The functional format is supported for compatibility.

ROUTINE

Macro Description

Purpose: To declare a routine.

Prototype:

```
label    ROUTINE   ENTRY,INTERNAL,EXTERNAL,
                   SAVE_AREA=,CALL_TYPE=,ONMSG=,
                   ONPGNT=,RSECT=,RSECT_LENGTH=,
                   PRNAME=,LINKAGE_TYPE=,PSECT=,
                   PARAMETER_TYPE=,ARTN=,SA_LENGTH=,
                   PSECT_LENGTH=,SYMBOL_PREFIX=,
                   GLOBAL_STORAGE_ADDR=,
                   GLOBAL_STORAGE_PTR=,NEW_STACK=,
                   INTERFACE_MACROS=,STACK_SLOP=,
                   NO_STACK_RC=
```

Parameters:

label — (required) is the name of the routine that is being declared.

SAVE_AREA — The name of the save area in the RSECT. The default is "xxxSA", where "xxx" are the first three characters of the RSECT name.

CALL_TYPE or PARAMETER_TYPE — The type of call made to this routine. "S" for S-form, "S(m,r)" or "S((min,max),r)" for S-form with m fixed arguments or between min and max variable arguments, and r return values. 0èrè1

"R(n,m)" for R-type with n arguments and m return values. Default is n = m = 0. Maximum m = maximum n = 10.

ONMSG — The name of the message on-unit to invoke to dispose of messages.

ONPGNT — The name of the PGNT on-unit to invoke if a PGNT occurs.

RSECT — The name of the RSECT. Default is the first three characters of the routine name followed by the string "RCT".

RSECT_LENGTH — The symbol to generate giving the length of the RSECT. Default is the first three characters of the routine name followed by the string "RCTL".

PRNAME — Specifies the name of the pseudo-register that is the transfer vector entry for the routine being declared. Usage of this option implies the loading of global storage from the second word of the pseudo-register entry.

ARTN — SPecifies where the routine address should be loaded from. The default is =A(label).

LINKAGE_TYPE — Specifies the linkage-type of the routine. Supported types are RM, CC, MTS, and OS. RM and CC are the same. This option controls what code is generated to enter, leave, and call the routine being declared. The default is CC.

PSECT=options — options is one of:

GPSECT(xxx,yyy) to call GPSECT with id xxx and length yyy to get space for global storage and the stack.

GETSPACE(len) or GETSPACE(bits,len) to call GETSPACE to allocate global storage and stack of length len (defaults to 2048). The second form allows a non-default switch value to be passed to GETSPACE (default is 3).

PARAM(n) indicates that parameter "n" in the S-type parameter list is a full word which points to the global storage and stack

HWIMB indicates that the Help-Where-Is-My-Buffer macro is to be used to find the MTS dsect for global storage.

MACRO(name ,par1,par2,...]) indicates that macro "name" is to be called to find the global storage. par1, par2, ... are

optional arguments to pass to the macro on the call.

The PSECT option is valid only for linkage-type OS routines.

PSECT_LENGTH — The length of global storage. This option is valid only for linkage-type OS routines, and indicates where in the allocated storage to draw the line between global storage and stack. The default is 8 bytes.

SA_LENGTH — Length of OS save area required by this routine when called. This option is usually used to allocate more than 72 bytes when updating the stack pointer in the stack descriptor before calling an OS routine.

SYMBOL_PREFIX — The prefix to use when generating symbols. The default is the first three characters of the routine name.

GLOBAL_STORAGE_ADDR=xxx
Specifies that "xxx" is the actually global storage for the routine. Valid only for linkage-type CC routines.

GLOBAL_STORAGE_PTR=xxx
Specifies that R11 should be loaded from location "xxx" before calling the routine. Valid only for linkage-type CC routines.

INTERFACE_MACROS — Specifies a parenthesized list of macros to be used to generate call, entry, and exit code for the specified routine. If any macro name is omitted, the corresponding default will be used.

STACK_SLOP — Specifies the size of the area to reserve at the end of the stack allocated upon entry to an OS type routine, to be used for detecting stack overflow. Defaults to 16*4, to allow a register save.

NO_STACK_RC — Specifies the return code to give in the event that the entry code failed to find/allocate a new stack. If this

parameter is omitted, the CCPUNT macro is called.

The following parameters may be specified as positional parameters:

INTERNAL	Designates an internal routine. This is the default.
ENTRY =name]	Designates an externally callable routine. If specified as ENTRY=name or ENTRY=(name1,name2,...,namen), the specified names are defined as aliases for the routine being declared.
EXTERNAL =name]	If specified as a simple positional parameter, EXTERNAL causes the generation of an assembler "EXTRN" statement for the routine. If specified as a keyword parameter (EXTERNAL=name), the name is used instead of the declared routine name.
NEW_STACK =len]	Specifies that the stack link in the stack descriptor should be used to find a new stack. If a length is provided, it means that if there is no next stack, one of length "len" should be allocated.
CLSTV	Specifies that the routine is to be called via the CLS transfer vector. The code to call the routine will assume that the first word of the global storage contains a pointer to the CLS transfer vector, and that a dsect with the name CLSTV has been defined. The routine name must be the same as the entry from CLSTV.

Description: The ROUTINE macro is probably the most important of all the macros, since it is the one that affects how all the others will work. The values specified on its invocation are saved in global symbols, and are used to do structure checking, and to determine what code should be generated.

Examples:

```
TEST    ROUTINE
```

This is the simplest possible invocation of the macro. It defines an internal routine TEST with no parameters, no result, and LINKAGE_TYPE=CC.

```
TEST    ROUTINE  EXTERNAL,PARAMETER_TYPE=R(2,1)
```

This example defines the routine TEST to have two R-type parameters (in R0 and R1), and one R-type result (in R0). It is externally callable, and has LINKAGE_TYPE=CC.

```
TEST    ROUTINE  LINKAGE_TYPE=OS,
                 PSECT=GPSECT(TEST,4096),
                 PARAMETER_TYPE=R(0,1)
```

This example defines the routine TEST to be an OS-linkage routine, that should call GPSECT to find the global storage and stack. Since the global storage size isn't specified, it defaults to 8 bytes. The routine has no parameters, and returns one result.

```
TEST    ROUTINE  LINKAGE_TYPE=OS,
                 PSECT=GPSECT(TEST,4096),
                 PARAMETER_TYPE=S(3,0)
```

This example defines the routine TEST to be an OS-linkage routine, that should call GPSECT to find the global storage and stack. Since the global storage size isn't specified, it defaults to 8 bytes. The routine has three S-type parameters, and no result.

SUBROUTINE

Macro Description

Purpose: To declare a subroutine.

Prototype:
```
label    SUBROUTINE   parent,PARAMETER_TYPE=,
                        CALL_TYPE=
```

Parameters:

label — (required) is the name of the subroutine that is being declared.

parent — is the name of the containing routine in which the subroutine appears. It must be declared by the ROUTINE macro before this macro call.

CALL_TYPE or PARAMETER_TYPE — are the same as for the ROUTINE macro.

Description: The SUBROUTINE macro declares a subroutine, which is a procedure which is internal to another routine. It saves registers and is self-addressable, but uses its parent routine's stack. Subroutines can call other routines, but they can only call other subroutines that belong to their parent routine.

Examples:

```
TEST     ROUTINE  ENTRY
TESTSUBRSUBROUTINETEST,CALL_TYPE=R(2,1)
```

This example defines the routine TEST, which is externally callable, and the subroutine TESTSUBR, which may only be called from within TEST. TESTSUBR takes two R-type parameters, and returns one R-type result.

RBEGIN

Macro Description

Purpose: To begin a routine.

Prototype: label RBEGIN LABEL=

Parameters:

label	(required) is the name of the routine being begun.
LABEL= YES×NO]	If LABEL=NO is specified, generation of the label for the routine name is suppressed. The default is LABEL=YES.

Description: The RBEGIN macro is a structure-delimiting macro. It indicates the beginning of a routine. It is illegal if already in a routine definition, unless RBEGINning a subroutine.

NOTE: The RBEGIN macro is not normally used explicitly, since it is emitted by the RENTER macro (q.v.). It is usually only required for routines that must have hand-coded entry sequences.

Examples:

```
TEST    RBEGIN
```

This example begins the routine TEST. No entry code is generated. TEST is defined as a label.

```
TEST    RBEGIN   LABEL=NO
```

This example begins the routine TEST, but does not define the label TEST. This is useful, for example, to begin a routine that is of the same name as the enclosing CSECT.

REND

Macro Description

Purpose: To end a routine, RSECT, pseudo-register, or pseudo-register vector definition.

Prototype: `label  REND  name,LEN=`

Parameters:

label	(optional) If ending a routine or pseudo-register vector, the statement label is ignored. If ending an RSECT or pseudo-register the statement label will be defined as the last+1 byte of the RSECT or pseudo-register.
name	(optional) is the name of the routine, RSECT, pseudo-register, or pseudo-register vector that is being ended.
LEN= YES×NO]	If LEN=YES is specified (the default), and an RSECT is being ended, REND will generate the EQU that defines the RSECT length symbol mentioned or defaulted in the routine declaration.

RENTER

Macro Description

Purpose: To begin a routine and enter it.

Prototype:

```
label    RENTER   LABEL=,OVERFLOW_EXIT=,
                  OVERFLOW_FLAG=
```

Parameters:

label	(optional) is the name of the routine being entered. If no label is given, RENTER assumes a routine has already been begun, and just generates the entry code.
LABEL= YES×NO]	If LABEL=NO is specified, generation of the label for the routine name is suppressed. The default is LABEL=YES.
OVERFLOW_EXIT=	is the label to branch to if a stack overflow occurs.
OVERFLOW_FLAG=	is the name of the flag to set if a stack overflow occurs. Only one of the OVERFLOW keywords may be specified. If OVERFLOW_CHECK=OFF was specified on the RSET macro, the OVERFLOW keywords are ignored.

Description: The RENTER macro is a structure-delimiting macro. It indicates the beginning of a routine. It is illegal if already in a routine definition, unless RENTERing a subroutine. RENTER emits a RBEGIN, followed by entry code appropriate to the LINKAGE_TYPE of the routine being entered.

Examples:

```
TEST    RENTER
```

This example enters the routine TEST. TEST is defined as a label.

```
TEST    RENTER   LABEL=NO
```

This example enters the routine TEST, but does not define the label TEST. This is useful, for example, to enter a routine that is of the same name as the enclosing CSECT.

```
TEST    RENTER   OVERFLOW_FLAG=OFLOW
```

This example enters the routine TEST. If a stack overflow occurs, the flag OFLOW is set to 1. OFLOW must be defined with the FLAGS macro (see MTS Volume 14).

RCALL

Macro Description

Purpose: To call a routine.

Prototype: `label RCALL routine,arg2,EXIT=,PARLOC=,VL=`

Parameters:

label	(optional) is a branch target.
routine	is the name of the routine being called, and possibly its parameter list (eg. routine(p1,p2,p3,p4)). The number of parameters must agree with the number that the routine was declared to have. Null parameters may be supplied to fulfill this requirement.
arg2	is not valid for calling an R-type routine. When calling an S-type routine, if the PARLOC keyword is not given, arg2 is the name of the parameter list, or if parenthesized, is the name of the register that points to the parameter list. If omitted, it is assumed that R1 already contains the address of the S-type parameter list.
EXIT=	is exactly the same as for the MTS READ, WRITE, etc macros, q.v.
PARLOC=	is not valid for calling an R-type routine. For calling an S-type routine, it specifies where the parameter list for the call should be built. If PARLOC is given, a parameter list may be concatenated to the routine name, as for R-type calls.
VL= YES×NO]	may be used to inhibit the setting of the VL-bit. The default is VL=YES.

Description: Legal forms of parameters and what they generate are ("Rx" means one of R0, R1, R2, or R3):

1. A simple name means a fullword:
   XXX generates L Rx,XXX
2. A parenthesized name means a register:
   (XXX) generates LR Rx,XXX
3. A quote-enclosed name means a character literal:
   'XXX' generates L Rx,=CL4'XXX'
4. A construct of the form A(...) means an address:
   A(XXX) generates LA Rx,XXX
   XXX may be a literal.
5. A construct of the form H(...) means a halfword:
   H(XXX) generates LH Rx,XXX
   XXX may be a literal.
6. A null parameter as in the list "x,,y" generates nothing -that is the value in the corresponding register is left unchanged.

Examples:

```
RCALL    TEST_R(=F'1')
RCALL    TEST_R(A(1))
```

These examples both call the R-type routine TEST, passing a 1 in R0.

```
RCALL    TEST_S(A(JUNK)),PARLOC=TESTPARS
```

This example calls the S-type routine TEST, passing the address of TESTPARS in R1. TESTPARS will contain the address of JUNK in the first word.

```
RCALL    TEST_S,TESTPARS
```

This example is similar to the previous one, except that the parameter list is assumed to have been previously constructed.

```
RCALL    TEST_S
```

This example is similar to the previous one, except that R1 is assumed to contain the address of an S-type parameter list.

RRETURN

Macro Description

Purpose: To return from a routine.

Prototype: `label RRETURN VALUE=,RETURN_CODE=`

Parameters:

label	(optional) is a branch target.
VALUE=	specifies the result(s) to be returned. It must be a parenthesized list with enough operands to satisfy the declaration. Null operands may be provided to fulfill this requirement. The operands have the save format as R-type parameters to the RCALL macro, q.v.
RETURN_CODE=	specifies the return code to be loaded into R15 before returning. If omitted, R15 is restored. The operand has the same format as R-type parameters to the RCALL macro, q.v. RC is a synonym for RETURN_CODE.

Examples:

```
RRETURN  VALUE=(,(R7))
```

This example returns to the caller, passing as results in R0 and R1 the current contents of R0 and R7. R15 is restored.

```
RRETURN  VALUE=(RESULT),RC=4
```

This example returns to the caller, passing the contents of the word at RESULT in R0, and giving a return code of 4 in R15.

REXIT

Macro Description

Purpose: To return from a routine and end it.

Prototype: `label REXIT VALUE=,RETURN_CODE=`

Parameters:

label	(optional) is a branch target.
VALUE=	specifies the result(s) to be returned. It must be a parenthesized list with enough operands to satisfy the declaration. Null operands may be provided to fulfill this requirement. The operands have the save format as R-type parameters to the RCALL macro, q.v.
RETURN_CODE=	specifies the return code to be loaded into R15 before returning. If omitted, R15 is restored. The operand has the same format as R-type parameters to the RCALL macro, q.v. RC is a synonym for RETURN_CODE.

Description: The REXIT macro just emits an RRETURN followed by an REND.

RSPARSET

Macro Description

Purpose: To build the parameter list for an S-type routine.

Prototype: label RSPARSET pars,routine,VL=

Parameters:

label	(optional) is a branch target.
pars	is the name of the location where the parameter list is to be built.
routine	is the name of the routine, concatenated to the parameter list that is to be built (eg. TEST(A(FDUB),A(JUNK))).
VL= YES×NO]	may be used to inhibit the setting of the VL-bit. The default is VL=YES.

Description: This macro is used to build parameter lists for S-type routines. It is normally used for parameter lists that don't change, but can also be used with the "routine,parlist" form of the RCALL macro, q.v.

Examples:

```
TEST_S  ROUTINE  PARAMETER_TYPE=S(2))
        .
        .
        RSPARSET TESTPARS,TEST_S(A(FDUB),A(JUNK))
        RCALL    TEST_S,TESTPARS
        .
        .
xxxRCT  RSECT
TESTPARSRSPARLOC TEST_S
        REND     xxxRCT
```

This example calls the S-type routine TEST_S. When TEST_S is entered, R1 will contain the address of TESTPARS. TESTPARS will

contain the address of FDUB in the first word and the address of JUNK in the second word.

RSPARLOC

Macro Description

Purpose: To reserve storage for a parameter list in an RSECT.

Prototype: label RSPARLOC routine,routine,...

Parameters:

label — (optional) is the name of the storage area that will be reserved.

routine,... — is a list of routine names. RSPARLOC will reserve a large enough block of storage for the largest parameter list.

Examples:

```
SERCOM   ROUTINE  EXTERNAL,LINKAGE_TYPE=OS,
                  PARAMETER_TYPE=S(4,1)
WRITE    ROUTINE  EXTERNAL,LINKAGE_TYPE=OS,
                  PARAMETER_TYPE=S(5,1)
         .
         .
TESRCT   RSECT
WRITPARSRSPARLOC  SERCOM,WRITE
         REND     TESRCT
```

This example will reserve five words in the RSECT.

RSECT

Macro Description

Purpose: To define an RSECT.

Prototype: label RSECT SA=

Parameters:

label	(optional) is the name of the RSECT that is being defined. It may be omitted if the RSECT is being defined within its parent routine.
SA= YES×NO]	If SA=YES (the default), a save area will be generated as the first 16 words of the RSECT.

Description: The RSECT macro is used to describe routine local storage. Its invocation is followed by DSECT-style definitions of any local variables that the routine needs. These variables can be automatically initialized by using the DCI macros. RSECTs are ended by the REND macro, q.v.

Examples:

```
TESRCT  RSECT
TES_SAVEDS      F                   Place to save a fullword
        REND    TESRCT
```

This example defines the RSECT TESRCT. It contains a 16-word save area at the front, followed by the fullword TESSAVE.

RSA

Macro Description

Purpose: To generate a save area.

Prototype: label RSA

Parameters:

label	(optional) the name of the save area. This must agree with the declared save area name for the containing RSECT's save area.

Description: This macro is normally generated automatically by the RSECT macro. It may be hand coded **only** if SA=NO was specified on the RSECT macro.

RDSECT

Macro Description

Purpose: To begin and end a DSECT.

Prototype: label RDSECT END]

Parameters:

label	(required) is the DSECT name to begin.
END	if specified, means that the dsect should be ended, and that a return should be made to the enclosing section.

Description: The RDSECT macro saves the current section when a DSECT is begun, so that it can be poped back to when the DSECT definition is completed. It is particularly useful in COPY sections.

Examples:

```
TEST    RDSECT
        .
        .
        RDSECT   END
```

REGS

Macro Description

Purpose: To generate register equates

Prototype: label REGS

Parameters: None

Description: This macro generates EQUs for the symbols R0-R15, FR0-FR6, NXTFRAME (R13), and CURFRAME (R12).

RRCTST

Macro Description

Purpose: Test a return code in a register.

Prototype: label RRCTEST list,R=

Parameters:

label	(optional) is a branch target.
list	is either a single branch target or a parenthesized list of branch targets. Alternatively, the targets may be specified as multiple simple positional operands. This form allows use of the expanded macro calling sequence, which in turn allows a comment to appear on each exit option.
R=	specifies the register to test. The default is R15 (of course).

Description: RRCTEST generates branching code in such a way that a return code higher than the highest expected value causes a branch to the location specified for the highest expected value. Null branch targets are translated into branches to the end of the return-code testing sequence.

Examples:

```
          RRCTEST  (A,B,,C)
```

In this example, a return code of 4 transfers to A, 8 to B, 12 causes no transfer, and 16 or greater transfers to C.

```
          RRCTEST  A,                  comment for RC 4
                   B,,                 comment for RC 8
                   C                   comment for RC 16
```

This example is just like the previous one, but uses the extended macro calling sequence so that a comment can appear for each return code.

RPRV

Macro Description

Purpose: To start a pseudo-register vector definition.

Prototype: label RPRV LCSPRCS=,ABSOLUTE=

Parameters:

label	(optional) is the name of the pseudo-register initialization routine. If omitted, no initialization routine is generated.
LCSPRCS=	(optional) is the name of the CSECT to put the LCSPR table in. If omitted, no LCSPR table is generated.
ABSOLUTE= NO×YES]	controls whether or not an LCSPR table with absolute as opposed to Q-con offsets is generated. The default is ABSOLUTE=NO.

Description: The RPRV macro begins the definition of a pseudo-register vector. See also RPRDEF, which is used to define pseudo-registers, and RTVENTRY, which is used to define transfer vector entries (which are, of course, special pseudo-registers).

RPRDEF

Macro Description

Purpose: To define a pseudo-register.

Prototype: `label  RPRDEF  alignment,length`

Parameters:

label	(optional) is the name of the pseudo-register being defined.
alignment	is either the keyword DSECT, which implies that a DSECT-type pseudo-register is being defined, or one of "D", "F", "H", or "B", for doubleword, fullword, halfword, or byte alignment, respectively, which implies that a DXD-type pseudo-register is being defined.
length	is the length of the DXD-type pseudo-register being defined.

Description: RPRDEF emits either a DXD instruction, or else emits a DSECT instruction and sets things up so that REND can end the DSECT. If the RPRDEF is inside a pseudo-register vector definition, RPRDEF either sets up so that REND can generate the LCSPR entry (for DSECT-type pseudo-registers) or generates the LCSPR entry (for DXD-type pseudo-registers).

Examples:

```
FAKEPR  RPRDEF   D,8
```

This example generates a pseudo-register with doubleword alignment and length 8.

```
FAKEPR  RPRDEF   DSECT
```

```
   FIELD1  DS       D                First field is a
                                     doubleword
   FIELD2  DS       F                Second is a fullword
           REND     FAKEPR           End the pseudo-register
                                     dsect
```

RTVENTRY

Macro Description

Purpose: To define a transfer vector entry.

Prototype: label RTVENTRY routine

Parameters:

label — (required) is the pseudo-register name for the transfer vector entry.

routine — (required) is the routine name.

Description: This macro is not normally coded by the programmer -- it is used by the ROUTINE macro to implement the PRNAME option. If the RTVENTRY macro appears inside a pseudo-register vector definition for which an initialization routine is being generated, code will be generated to initialize the transfer vector entry.

Examples:

```
PRGTSPCERTVENTRY UCGTSPCE
```

This example defines the pseudo-register PRGTSPCE as two fullwords. The first will contain the address of UCGTSPCE, and the second will contain the value that UCGTSPCE expects to be in R11 when it is called.

RPR

Macro Description

Purpose: To generate an RX- or RS-format instruction that references a pseudo-register or a location in a pseudo-register DSECT.

Prototype: label RPR inst,opnd1,opnd2,opnd3,COMMENT=

Parameters:

label	(optional) is a branch target.
inst	is the instruction to emit.
opnd1,opnd2 ,opnd3]	are the operands for the instruction. The syntax of the operands follows the normal assembler rules, with the additional requirement that if a symbol is a component of a DSECT that describes a pseudo-register, it must be coded as ".SSS", where "SSS" is the actual symbol. Similarly, if a pseudo-register is being referenced directly, its name must be suffixed with a ".", i.e. it must be coded in the form "DDD.".
COMMENT=	is a comment to emit with the instruction.

Examples:

```
          RPR      L,R1,FAKEPR.
```

This example loads R1 with the contents of the pseudo-register FAKEPR.

```
          RPR      L,R15,.FIELD1
```

This example loads R1 with the contents of the field FIELD1, which is a member of a DSECT-type pseudo-register.

RPRI

Macro Description

Purpose: To generate an SI-format instruction that references a pseudo-register or a location in a pseudo-register DSECT.

Prototype: `label  RPRI  inst,opnd1,opnd2`

Parameters:

label	(optional) is a branch target.
inst	is the instruction to emit.
opnd1,opnd2	are the operands for the instruction. The syntax of the operands follows the normal assembler rules, with the additional requirement that if a symbol is a component of a DSECT that describes a pseudo-register, it must be coded as ".SSS", where "SSS" is the actual symbol. Similarly, if a pseudo-register is being referenced directly, its name must be suffixed with a ".", i.e. it must be coded in the form "DDD.".

Examples:

```
          RPRI     TM,FAKEPR.,X'01'
```

This example executes a TM instruction on the contents of the pseudo-register FAKEPR.

RPRC

Macro Description

Purpose: To generate an SS-format instruction that references a() pseudo-register(s) or a() location(s) in a pseudo-register DSECT.

Prototype: `label  RPRC  inst,opnd1,opnd2`

Parameters:

label	(optional) is a branch target.
inst	is the instruction to emit.
opnd1,opnd2	are the operands for the instruction. The syntax of the operands follows the normal assembler rules, with the additional requirement that if a symbol is a component of a DSECT that describes a pseudo-register, it must be coded as ".SSS", where "SSS" is the actual symbol. Similarly, if a pseudo-register is being referenced directly, its name must be suffixed with a ".", i.e. it must be coded in the form "DDD.".

Examples:

```
          RPRC     CLC,0(4,R1),FAKEPR.
```

This example compares the location pointed to by R1 with the contents of the pseudo-register FAKEPR.

RNEXT STACK

Macro Description

Purpose: To switch to the next stack in the stack of stacks, and set up R12 and R13 appropriately.

Prototype: label RNEXT_STACK

Parameters:

label	(optional) is a branch target.
FULLSAVE= YES×NO]	If FULLSAVE=YES is specified, than the registers are saved into the new stack after the switch is made. The default is FULLSAVE=YES.

Description: This macro is primarly designed for use in exit routines, where some initial processing must be done before the stack switch can be made (since the stack switch can't be done as part of the entry sequence).

Examples:

```
RNEXT_STACK
```

This example switches to the next available stack. If there is no available stack, the CCPUNT macro is called. After switching to the next stack, R0 through R15 are saved in the save area of the routine's RSECT.

RPREV STACK

Macro Description

Purpose: To revert to the previous stack in the stack of stacks.

Prototype: label RPREV_STACK

Parameters:

label (optional) is a branch target.

Description: This macro is primarly designed for use in exit routines, to return the stack descriptor to its previous state, immediately before doing a POPQ.

Examples:

```
RPREV_STACK
```

This example switches back to the previous stack in the stack of stacks.

RPUSH

Macro Description

Purpose: To push a state variable onto an assembly-time stack.

Prototype: label RPUSH option, option,...]

Parameters:

label — the label field is ignored.

option — is either the keyword SECTION, or else any legal ASMH PUSH pseudo-op operand.

Description: The RPUSH macro is provided because stupid ASMH can't PUSH and POP the current section.

Examples:

```
          RPUSH    SECTION,USING
```

This example saves the current section and using state.

RPOP

Macro Description

Purpose: To pop a state variable off an assembly-time stack and make it the current state.

Prototype: label RPOP option, option,...]

Parameters:

label — the label field is ignored.

option — is either the keyword SECTION, or else any legal ASMH POP pseudo-op operand.

Description: The RPOP macro is provided because stupid ASMH can't PUSH and POP the current section.

Examples:

```
          RPOP     SECTION,USING
```

This example restores the previous section and using states.

RPUSHSTACK

Macro Description

Purpose: To grab space at the end of the current stack frame.

Prototype: `label  RPUSHSTACK  addr,len,EXIT=,ERRFLAG=,R=`

Parameters:

- label — (optional) is a branch target.
- addr — is the location of a fullword, or a register in parentheses, in which the current stack top pointer can be saved.
- len — is the number of bytes to grab.
- EXIT= — is the label to branch to if a stack overflow occurs.
- ERRFLAG= — is the name of the flag to set if a stack overflow occurs.

  Only one of the EXIT or ERRFLAG keywords may be specified. If OVERFLOW_CHECK=OFF was specified on the RSET macro, they are ignored.
- R= — If either the EXIT or ERRFLAG keywords is specified, a temporary register (defaults to R15) will be used to make the stack overflow check.

Description: This macro is used to grab space at the end of the current stack frame. This is done by bumping the value in R13. RPUSHSTACK and RPOPSTACK can be used together to make sure that a certain amount of space exists at the end of the stack, before calling some other routine that doesn't do its own stack checking.

Examples:

```
          RPUSHSTACKSAVER13,256
```

This macro increments R13 by 256. No stack overflow checking is done.

```
          SET       STK_OFLO,OFF
          RPUSHSTACKSAVER13,256,ERRFLAG=STK_OFLO
          RPOPSTACKSAVER13
```

This example checks to make sure that there are at least 256 bytes available at the end of the stack. If not, the flag STK_OFLO is set TRUE.

RPOPSTACK

Macro Description

Purpose: To pop the stack end back to the location saved by a previous RPUSHSTACK.

Prototype: label RPOPSTACK addr

Parameters:

label	(optional) is a branch target.
addr	is the location of a fullword, or a register number in parentheses, to which the current stack top pointer should be restored.

Examples:

```
RPOPSTACK(R5)
```

In this example, the stack top pointer is restored from R5.

```
RPOPSTACKSAVER13
```

In this example, the stack top pointer is restored from the fullword SAVER13.

RSET

Macro Description

Purpose: To set various options that control how the rest of the macros function.

Prototype: label RSET STACK_OVERFLOW_CHECK=,OLD_PR_REF=

Parameters:

label — the label field is ignored.

STACK_OVERFLOW_CHECK= YES×NO] — if NO, specifies that no stack overflow checking is to be done, even if it is requested on the RENTER macro. The default is STACK_OVERFLOW_CHECK=YES, but the checking must be requested on the RENTER macro in order to have the checking code be generated.

OLD_PR_REF= NO×YES] — if YES, specifies that DSECT-type pseudo-register references will be coded as "DDD.SSS", where DDD is the dsect name and SSS is the symbol name. The default is OLD_PR_REF=NO. The "DDD.SSS" notation is the old ASMG notation. If used, qualified symbols (named USINGs) will not be available on any operand that can possibly be a pseudo-register reference, and the assembly will take longer.

RDISPLAY

Macro Description

Purpose: To display the attributes of a routine.

Prototype: label RDISPLAY rtn

Parameters:

label	the label field is ignored.
rtn	is name of the routine to display.

Description: In addition to being the name of the routine to display, "rtn" may be one of the following special items:

1. "(X)" displays the routine that &RTN points to, if any.
2. "(NON)" displays the current non-trivial routine, if any.
3. "(SUBR)" displays the current subroutine, if any.
4. "(CURRENT)" displays the current routine, if any.
5. "(RSECT)" displays the routine for the current RSECT, if any.

CCPUNT

Macro Description

Purpose: To punt after an error is detected by the macros.

Prototype: label CCPUNT

Parameters:

label (optional) is a branch target.

Description: This macro is invoked by various other macros when they detect an error condition. It generates a DC H'0'. It is provided so that individual programs can replace it with something more meaningful, if desired.

VI. Differences From Earlier Versions

The current version of the coding conventions is the third main version since the coding conventions were originally designed, although there have been ongoing minor changes. Differences between these versions are described here.

A. DIFFERENCES BETWEEN VERSION 3 AND VERSION 2

1. Stack overflow checking

Stack overflow checking is implemented through the use of stack descriptors. For each stack, there is a six-word stack descriptor (located, by convention, as the six-words immediately before the stack, and allocated as part of the stack allocation. Thus, a 4096 byte stack actually has only 4072 usable bytes.) The words in this descriptor are:

1. The upper limit of the stack (highest address).
2. The current stack pointer. This word is set as part of the process of calling a non-coding conventions routine, and is thus valid **only** when not in the coding conventions environment. It is used when re-entering the coding conventions environment, to re-establish the stack.
3. The stack base (lowest address).
4. The address of the next stack descriptor (or zero).
5. The address of the previous stack descriptor (or zero).
6. Unused (always zero).

The next and previous pointers are used to implement a "stack of stacks", so that exit routines or other asynchronous code can use a fresh stack without fear of corrupting the current stack.

The six-word stack descriptor described above is called the **full** stack descriptor. In addition, there is a **global** stack descriptor, which is four words long. The address of the global stack descriptor is assumed to be at offset 4 in the global storage (4(,R11)). The global stack descriptor always describes the **current** stack. The first three words of the global stack descriptor are the same as the first three words of the full stack descriptor. The fourth word of the global stack descriptor is the address of the full stack descriptor for the current stack.

So, with all that out of the way, stack overflow checking is implemented by comparing the R13 value at entry to a routine with the stack limit word in the global stack

descriptor.

2. Global Storage Switching

In earlier versions of the coding conventions, there was only one block of global storage, and it was global to the entire set of programs that were making use of the coding conventions. This made it necessary, for example, for **all** Plus programs to allocate sufficient global storage to account for the sub-tasking monitor's global storage, just in case **any** Plus program should want to use the sub-tasking monitor. To avoid this problem, the concept of global storage switching was invented. In this concept, a package of subroutines (say, the sub-tasking-monitor), may want to have an area of global storage. The initialization routine for the package will allocate this storage, and return a pointer to it. On all subsequent calls to routines in the package, the caller is expected to provide this address in R11.

3. Replacement of trivial routines

Versions 1 and 2 of the coding conventions had two different types of routines: non-trivial and trivial. Experience with trivial routines showed that they were too trivial to be very useful. The fact that they didn't save and restore registers and the fact that they couldn't call each other have proved to be impractically severe restrictions.

On the other hand, there was a need for the ability to have several routines share a single RSECT and set of working register values, as trivial routines and their parent non-trivial routines did.

Accordingly, it was decided to implement a new type of routine called a SUBROUTINE that has some of the properties of a trivial routine without all of its limitations. A subroutine "belongs" to a routine in the sense that it is coded physically inside the routine and shares the routine's RSECT. However, each subroutine saves and restores registers (on the stack, just after the end of the current RSECT), and each subroutine is self-addressable using R10. Subroutines are called by BALR R14,R15 and may call each other as long as all subroutine-subroutine calls do not cross from one routine's subroutine set into another's.

Any routine declared to be a TRIVIAL routine will be converted (with a message) to a subroutine by the current coding conventions.

4. General cleanup

Compatible modifications were made to take advantage of some ASMH features. These changes improved the quality of the code in the macros (as much as cleanup is possible in so ghastly a language) and increased the speed of doing assemblies using them.

In addition to the above changes, a number of minor internal and external improvements were made in the process of making the update.

B. DIFFERENCES BETWEEN VERSION 2 AND VERSION 1

1. Reversing the roles of R12 and R13

In version 1 of the coding conventions, R13 was used as the current stack frame base, and R12 as the next stack frame base. This meant that R13 had to be set before any call to an OS routine, and then restored when that routine returned. It was observed that by simply reversing the usage of R12 and R13, this problem could be avoided.

2. Removal of scratch registers

The original idea of having R0 through R3 not be restored by routine calls was not well-received by programmers outside UBC. Accordingly, the macros were changed to restore all registers (except for return values, of course) after each call.

3. Making R11 the MTS dsect base in the MTS assemblies

Originally, R4 was made the MTS DSECT (i.e. global storage) base register in the MTS usage of the coding conventions because R4 has traditionally been the MTS DSECT base register in pre- coding conventions code. However, it was agreed that R11 is a more convenient register to use for this purpose. This is partly because the general coding conventions use R11 for their global storage pointer, so that using R11 in the MTS conventions would make both standards the same in this area. More importantly, R11 is more convenient because R4 is in the middle of the working registers, while R11 is at the upper end of the register set, where all of the other reserved registers are.

Making this change allowed the MTS assemblies to use the standard macros, with the extensions described in the next section.

VII. The Coding Conventions and the MTS assemblies

Use of the coding conventions in the MTS assemblies is basically the same as their use elsewhere, except for a few details, which will be described here.

A. INTERFACING MACROS

All the usual macros are used by the MTS assemblies. In addition, two other macros are provided to interface between routines coded according to the conventions and routines coded not according to the conventions. These macros are RICALL, to allow an old-style, non-coding conventions routine to call a coding conventions one, and RAENTER, to define an interface routine that lets a coding conventions routine call an old-style non-coding conventions one.

RICALL is a macro that sets up the coding conventions environment, then does an RCALL to the routine being invoked. It is used just like RCALL, except that it must be coded **outside** a declared routine, while RCALL must be coded **inside** one.

RAENTER is a macro that is callable from the coding conventions environment, but which sets up the old-style MTS environment before it finishes. The purpose of RAENTER is to facilitate the coding of what are called **adaptor routines.** An adaptor routine is an interface that surrounds an old-style routine and makes it look like a new-style one. The form of an adaptor routine is:

```
name    RAENTER
        .                           (code to call the
        .                           old-style routine)
        RRETURN                     (as usual)
        REND                        (as usual)

        RSECT                       (as usual)
        REND                        (as usual)
```

An adaptor routine is declared and called just as if it were a normal new-style routine.

The reason for having adaptor routines the way they are is that, if and when all direct calls to the old-style routine are done away with, the adaptor routine can simply be replaced by a real new-style routine that does the same thing as the old-style one did. Thus, the adaptor routine principle provides an evolutionary path from old-style code to new-style code.

INDEX