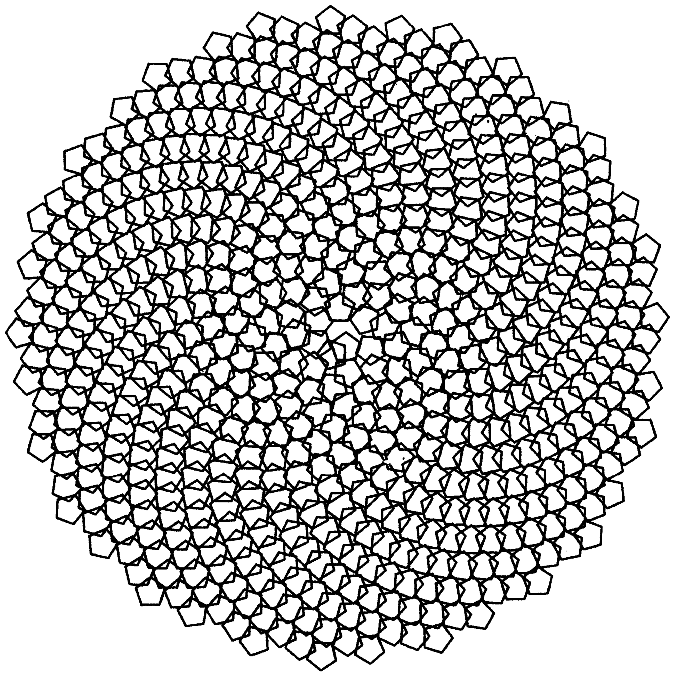
GDDM

REXX Guide

IBM


Front Cover Pattern: Electronic Sunflower

SC33-0478-0
Release 1

G D D M

REXX Guide

Program Number
GDDM-REXX 5664-336

Release 1

Licensed Program

IBM

First Edition (January 1987)

This edition applies to Release 1 of the GDDM-REXX IBM licensed program, program number 5664-336.

"GDDM" stands for "Graphical Data Display Manager"; "REXX" stands for "Restructured Extended Executor Language."

Changes are made periodically to the information herein; before using this publication, consult the latest *IBM System/370, 30xx, and 4300 Processors Bibliography*, GC20-0001, to learn which editions and technical newsletters are current and applicable.

Publications are not stocked at the addresses given below; requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed either to:

International Business Machines Corporation,
Department 6R1H,
180 Kost Road,
Mechanicsburg,
PA 17055,
USA

or to:

IBM United Kingdom Laboratories Limited,
Information Development, Mail Point 095,
Hursley Park,
Winchester, Hampshire, England SO21 2JN

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Preface

What this book is about

This book describes GDDM-REXX, a programming productivity tool that lets GDDM be used from EXECs written for the VM/System Product Interpreter.

Who this book is for

This book is for users of GDDM-REXX and those who install it.

What you need to know

This book assumes some knowledge of high-level language programming and of CMS.

How to use this book

Introductory information and learning sessions are at the front of the book, installation and diagnosis information in the middle, and reference at the back. Turn to the section you are interested in. Note that for reference information on REXX statements and GDDM calls you will need other books (see the Bibliography that follows).

GDDM books

Introduction

- General Information — GBOF-0058*
- Release Guide — GC33-0320

*Includes the GDDM brochures. For the General Information manual only, use order number GC33-0319

General

- Installation and System Management for MVS — SC33-0321
- Installation and System Management for VM — SC33-0323
- Installation and System Management for VSE — SC33-0322
- GDDM-GKS Installation Guide — SC33-0439
- Performance Guide — SC33-0324
- Messages — SC33-0325
- GDDM-GKS Messages — SC33-0496
- Diagnosis and Problem Determination Guide — SC33-0326

Programming

- Application Programming Guide (Two volumes) — SC33-0337
- Base Programming Reference (Two volumes) — SC33-0332
- GDDM-PGF Programming Reference — SC33-0333
- GDDM-REXX Guide — SC33-0478 — this book
- GDDM-GKS Programming Guide and Reference — SC33-0334
- Base Programming Summary (Booklet) — SX33-6053
- GDDM-PGF Programming Summary (Booklet) — SX33-6054

User's Guides

- Guide for Users — SC33-0327
- Interactive Chart Utility (ICU) — SC33-0328
- Image Symbol Editor — SC33-0329
- Vector Symbol Editor — SC33-0330
- Interactive Map Definition (GDDM-IMD) — SC33-0338

VM/System Product Interpreter books

VM/System Product Interpreter User's Guide, SC24-5238

VM/System Product Interpreter Reference, SC24-5239

VM installation and system books

VM/SP Planning Guide and Reference, SC19-6201

VM/SP Operator's Guide, SC19-6202

VM/SP System Programmer's Guide, SC19-6203

VM/SP System Messages and Codes, SC19-6204

VM/SP Installation Guide, SC24-5237

Book structure

Contents

Part 1: Learning

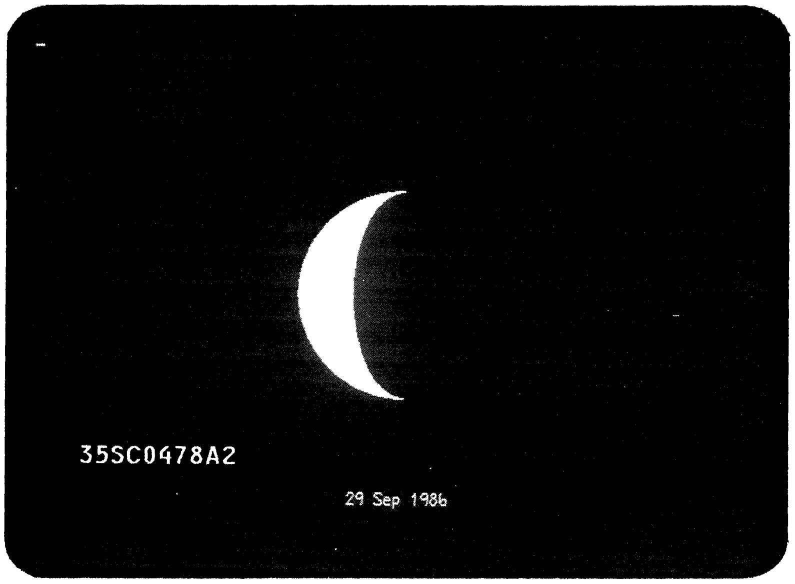
```
/* REXX EXEC to draw phases of the moon                          */
Address command 'GDDMREXX INIT'    /* initialize GDDM-REXX        */
Address gddm                       /* tell REXX to pass calls to GDDM*/

period = 29.5306                   /* time between new moons      */
quarter = period/4                 /* quarter period of rotation  */
day = date('C')-31421              /* number of days since first new */
                                   /*    moon in 1986 (10th January) */
tdate = date()                     /* today's date in character form */
phase = day//period                /* number of days after new moon  */
rightrad = 20                      /* radius of right edge of moon   */
leftrad = 20                       /*    and left edge to be drawn   */
inc = leftrad/(period/4)           /* assume phase changes regularly */

Select                             /* select shape of moon according */
                                   /* to which quarter we are in     */
   When phase<quarter then         /* first quarter                  */
      leftrad=-(leftrad-(phase*inc))
   When phase<quarter*2 then       /* second quarter                 */
      leftrad=((phase-quarter)*inc)
   When phase<quarter*3 then       /* third quarter                  */
      rightrad=rightrad-((phase-quarter*2)*inc)
   Otherwise                       /* fourth quarter                 */
      rightrad=-((phase-quarter*3)*inc)
End

                                   /* use GDDM to draw moon with two */
                                   /* elliptical arcs, direction     */
                                   /* varies depending on the quarter*/
'GSUWIN 0 100 0 100'               /* set up grid 100 by 100         */
'GSCOL 7'                          /* color 7 is white               */
'GSMOVE 50 30'                     /* move to start of arc           */
'GSAREA 1'                         /* area to shade moon             */
'GSELPS 20 .rightrad 90 50 70'     /* draws right hand edge          */
'GSELPS 20 .leftrad 90 50 30'      /* draws left hand edge           */
'GSENDA'                           /* end the area                   */
'GSCHAR 40 10 . .tdate'            /* write date under moon          */
'ASREAD . . .'                     /* send result to the terminal    */

Address command 'GDDMREXX TERM'    /* close down GDDM-REXX           */
Exit                               /* terminate the program          */
```

Introduction to GDDM-REXX

GDDM-REXX lets you use GDDM in EXECs written for the VM/System Product Interpreter, using the Restructured Extended Executor Language – REXX. On the opposite page is a listing of a GDDM-REXX program. The program draws a picture of the phase of the moon on the day you run it.

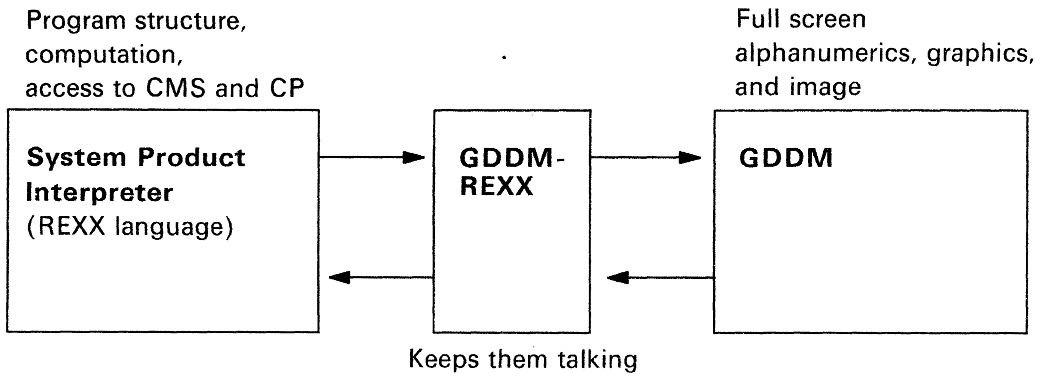
A look at the program shows you how easy it is to use GDDM-REXX. The REXX language is easy to code and is uncluttered; it uses high-level language statements such as SELECT and IF...THEN...ELSE. REXX also lets you use CMS and CP system facilities. GDDM calls can be coded using REXX variables or literals as parameters. REXX programs are interpreted – that means you do not have to go through the chore of compiling and recompiling the program. You can code and run, then change and run again until you have your program exactly how you want it. These features let you write and test your programs quickly.

GDDM-REXX is an IBM licensed program; prerequisites are the VM/System Product Release 4 or later and GDDM/VM Version 2. If you have installed the GDDM-PGF or GDDM-GKS licensed program, you will be able to use them with GDDM-REXX. A description of prerequisites is given in the installation section of this manual.



Background to GDDM-REXX

The diagram below shows the relationship between GDDM, GDDM-REXX, and the System Product Interpreter, which is the program that interprets programs in the REXX language.


Program structure, computation, access to CMS and CP
Full screen alphanumerics, graphics, and image
System Product Interpreter (REXX language)
GDDM-REXX
GDDM
Keeps them talking

A brief description of all three follows on the next few pages under the headings "Three-page REXX," "Three-page GDDM," and "Three-page GDDM-REXX."

If you are new to GDDM and REXX, you may find it hard to know which you are dealing with at any particular point in a program. Here is a brief explanation: GDDM lets you format graphics and alphanumerics so that the results you want can be shown on a screen or printed or plotted on paper; REXX is a high-level programming language – it lets you write a program that contains the GDDM calls and perhaps computes or generates the information that GDDM will display.

Here is a sequence of statements that shows the principle:

```
/* series of REXX statements and GDDM calls                         */
x = x+y                         /* REXX computation statement       */
'GSLINE .x .y'                  /* GDDM graphics call               */
'ASREAD .typ .val .count'       /* GDDM output statement            */
If typ = 1 & val = 3 then       /* REXX conditional statement       */
   Exit
Else                            /* REXX conditional statement       */
   Call newpic                  /*    and call                      */
```

In the example, GDDM calls are in uppercase letters and within quotes, and REXX statements are in lowercase, with initial capitals on statement names. This is a convention used in this book for clarity. You may like to use it yourself, or you may have a preferred alternative style. However, the important difference lies in the different jobs that GDDM and REXX are used for.

Three-page REXX

This description is for people who already know a high-level language. If you know nothing of programming you will need more than the information that is here. Full descriptions of REXX may be found in the *VM/System Product Interpreter User's Guide*, SC24-5238, and the *VM/System Product Interpreter Reference*, SC24-5239.

REXX is a powerful programming language that is used in the System Product Interpreter on the VM system. It is easy to code because it includes high-level language features such as IF...THEN...ELSE, SELECT, DO loops and others. It treats data variables as being typeless, that is, their attributes are determined dynamically, and so there is no problem of data declaration or definition that takes up so much programming space in many other high-level languages. Another reason why REXX is easy to code is that it gives easy access to other environments; for example, CMS or CP commands can be coded directly into REXX programs.

Example of a REXX program: Here is a simple REXX program that shows the powers of four; four to the power of 0, four to the power of 1, and so on:

```
/* program to show exponentiation                                */
x=4                               /* show the powers of four     */
Do i=0 to 5                       /* 4 to the power of 0 to 5    */
   y=x**i                         /* y = x to the power of i     */
   Say x 'to the power of' i '=' y /* show result on terminal    */
End i                             /* end of the do loop          */
Exit                              /* end of the program          */
```

The program starts with a comment, as all REXX programs must. (REXX comments start with a `/*` and end with a `*/`.) Each statement is on one line with no end marker. (No end marker is needed when there is only one statement per line.) If two statements occur on a line, a semicolon is used as a delimiter; if a statement overflows a line, a comma is used as a continuation character. The program ends with `Exit`. The words in the `Say` statement are in single quotes. This program is in uppercase and lowercase characters, but it need not have been – more of that later.

How programs are held and executed: In CMS, REXX programs are normally produced as files with a filetype of EXEC. They are invoked by typing their name as a command or by using the EXEC command.

Getting input into REXX: REXX can get its input from a stream or a stack, from reading files using the EXECIO command, or from arguments passed to the program.

When using a stack, REXX pulls the input off the stack, parses it and moves it into variables. This stack can be the terminal. Thus the loop below will get two numbers from the terminal until `STOP` is typed.

```
/* get values until STOP is typed                                */
Do until number1='STOP' | number2='STOP'
   Say 'Type in two numbers'
   Pull number1 number2
End
```

If there is no second number, the variable **number2** will have a null value assigned to it; if you type in more than two numbers, **number2** will contain everything after the first. This will only cause a problem if you try to use it in an arithmetic operation like adding.

Arguments to REXX programs: Arguments to a REXX program can be got by use of the **Arg** instruction thus:

```
/* rexx program with two arguments                                    */
Arg arg1 arg2
Say 'The two arguments are ' arg1 'and' arg2
```

You might prefer to use the **Parse arg** instruction instead; it is more flexible and more powerful.

The exponentiation example on the previous page could be improved by the use of arguments:

```
/* program to show exponentiation                                     */
Arg x                           /* read the argument                  */
```

If the program was called POWERS you could then type in **POWERS 5** to get the powers of 5 – you would have to remove the assignment **x=4**.

Subroutines and functions: Subroutines are delimited by labels (like **adder** in the following example) and **Return** statements and are called by **Call** statements.

```
/* REXX program with two arguments                                    */
   Call adder 1 2
   Exit                         /* end of mainline code               */
adder:
   Arg num1 num2
   Say num1 + num2
   Return
```

Notice that the subroutine comes after the **Exit** statement that ends the main part of the EXEC. Subroutines can also be programs held in separate files.

You can also write your own functions – subroutines that return values, like this one, to calculate factorials:

```
/* example of a recursive internal function                           */
   Arg x
   Say  x'! =' factorial(x)
   Exit

factorial: Procedure
   Arg n
   If n = 0 then
      Return 1
   Return factorial(n-1) * n
```

Arrays and stemmed variables: REXX has extended the high-level language idea of arrays. The extensions are variables with stems. Stemmed variables can have numbered subscripts which can be accessed in do-loops just like arrays; for example:

```
Do i = 1 to 10
   Say stem.i
End i
```

Notice the subscript is referenced by a dot (a period) followed by a number. Multi-dimensional arrays use a repeat of the dot, `array.1.1` and so on. To give a value of 6 to every element of an array use:

```
array.=6
```

Also, the subscripts need not be numbers. `array.joe` is just as valid as `array.1`, even if `joe` is a character string. In addition, the existence of `array.10` does not mean there has to be an `array.9`.

For a programmer used to other high-level languages, a simple way to look at it is that REXX uses the item after the stem as a suffix, which can, but need not, act as a subscript.

Uppercase and lowercase: REXX programs tend to be written in a combination of uppercase and lowercase not unlike English prose. Variables are given the value of their name in uppercase if they do not have a previously defined value. (That is why if you type `Say hello`, the terminal displays `HELLO`.)

If you want REXX to keep something in lowercase you must enclose it in quotes (single or double):

```
words='Hello there'
```

To prevent input from being translated into uppercase you use `Parse pull` explicitly instead of `Pull` which is short for `Parse upper pull`.

Quotes and double quotes: REXX gives you the choice of two sorts of quotes, single and double. You don't need to double them up to put a quote in a string (as in some programming languages) unless the quote is already used as a delimiter:

```
Say "It's easy to get quotes in a string"
Say 'It''s easy this way, too'
```

To the newcomer, a REXX program looks full of quotes. Because REXX has no data declaration or definition, anything may be a variable name and potentially may be replaced by a value. For this reason everything that is not a variable, such as a GDDM call name or a CP or CMS command, is safer if it is included in quotes.

REXX expressions: REXX has a powerful collection of instructions (like `Say`) and built-in functions (like `date`). Any of these can contain expressions, which are evaluated, in a manner similar to other programming languages. Any expression that is not a REXX instruction or assignment is treated as a command to be processed by some external environment.

By default, commands are passed to CMS for execution. However, you can establish a different environment for them using the `Address` instruction. This tells REXX to pass the command elsewhere, and is the way GDDM-REXX receives GDDM calls. A command like `'GSLINE 30 40'` in a REXX EXEC is passed to GDDM-REXX for processing by GDDM.

Three-page GDDM

GDDM is a program that lets you show graphics, alphanumerics, and images on a screen and lets you print or plot the result.

This description gives a very brief overview. It will tell you enough to get started using GDDM-REXX; for serious GDDM programming you will need to use the other GDDM manuals.

The *GDDM Application Programming Guide* is the most useful book if you want to learn to program with GDDM. You will also need other reference manuals, listed on page 19; order numbers are on page iv at the front of this book.

Also, the ERXPROTO EXEC, supplied with GDDM-REXX, should help you code many of the GDDM calls — more about this later.

Input and Output: When a graphics call or an alphanumerics call is executed, the drawing or writing is done on a conceptual screen held by GDDM. The output is not sent to the terminal until one of the following input/output calls is executed:

`ASREAD`	Normal for terminal
`FSFRCE`	For printers and terminal "animation"
`GSREAD`	Special purpose, interactive graphics
`MSREAD`	Special purpose, mapping
`WSIO`	Special purpose, operator windowing

Alphanumerics: GDDM has two forms of alphanumerics: procedural alphanumerics, which is suitable for small applications, and mapped alphanumerics, which is more suitable for large menu-based systems like order processing, and provides better performance.

Procedural alphanumerics lets you define a field that is within an area on the screen, and give the field an identifying number so you can refer to it again. You can then read data from or write data into the field. For example:

```
/* set up values in REXX                                          */
id=1                              /* identifier for field 1       */
row=3                             /* starts on row 3...           */
col=1                             /* ...in column 1               */
depth=1                           /* 1 row deep                   */
width=80                          /* 80 columns wide              */
type=0                            /* takes input and output       */
'ASDFLD .id .row .col .depth .width .type' /* create the field    */
'ASCPUT .id 26 "These words are in a field" ' /* put words in field */
'ASREAD .typ .val .count'         /* send to screen               */
'ASCGET .id 80 .inputval'         /* put input in inputval        */
```

You can control the color and other attributes of complete fields or of individual characters within the field. Remember that the words will not appear on the screen until you use an input/output statement.

Mapped alphanumerics lets you create a template or "map" of the screen which is separate from the program. The map associates positions on the screen with variable names. You then put values in the variables, call on the map to draw the screen, and return input into the variables. The map is created with a GDDM utility, GDDM Interactive Map Definition. (GDDM-IMD is a separate licensed program.) The calls involved are:

MSPCRT	Create a mapped page
MSDFLD	Define mapped field on page
MSPUT	Put values on the mapped screen
MSGET	Get values from the mapped screen
MSREAD	A combination of MSPUT, MSGET, and ASREAD

Graphics: This is produced by calls that draw on a two-dimensional grid. Calls to draw lines are given an end point and draw from the current position to the end point. You can move the current position with GSMOVE, and draw straight lines with GSLINE. The grid is 100 by 100 unless you change it. The calls below draw a triangle near the middle of the screen. ASREAD sends the drawing to the terminal and waits for you to press an action key.

```
/* program to draw a triangle                                        */
Address command 'GDDMREXX INIT'   /* initialize                      */
Address gddm                      /* pass commands to GDDM           */
'GSMOVE 50 50'                    /* move to point at X=50 Y=50      */
'GSLINE 70 50'                    /* draw line to point X=70 Y=50    */
'GSLINE 70 70'                    /* draw line to point X=70 Y=70    */
'GSLINE 50 50'                    /* draw line to point X=50 Y=50    */
'ASREAD .typ .val .count'         /* send drawn triangle to screen   */
Address command 'GDDMREXX TERM'   /* terminate                       */
Exit                              /* stop the program                */
```

As well as straight lines, there are calls to draw elliptical or circular arcs, and to define areas. With these it is possible to draw almost anything.

The color and width of the lines are set by attribute calls. For example, you can say that all lines are to be yellow (color 6) by the call

```
'GSCOL 6'                         /* following graphics in yellow    */
```

Annotation of pictures is done with *graphics text.* This lets you write notes on the picture. For example:

```
'GSCHAR 50 45 8 "Triangle"'       /* write Triangle, 8 characters    */
                                  /* starting at X=50 Y=45           */
                                  /* Note use of "" round strings    */
```

Other things you can do with graphics calls are control the grid and the size of the picture, and divide pictures into segments (repeatable elements like the windows for a house). Note that the picture will not appear on the screen until you use an input/output statement such as ASREAD.

Charts and graphs: GDDM-PGF, one of the programs in the GDDM family, is for drawing charts. It provides two methods, the Interactive Chart Utility (ICU) and Presentation Graphics Routine calls, that let you build your own charts. The ICU can be used by non-programmers to draw charts. The calls are a programming interface. The ICU can be used on its own or can be called from a program. Calling the ICU from a program can be useful for tasks such as printing or displaying charts whose values are held in files. GDDM-PGF is a separate licensed program and must be installed before the charting calls can be used.

Interactive graphics: GDDM has a series of interactive graphics calls that pass a screen position to the program and allows the picture to be moved or changed from the display terminal.

Image handling: GDDM handles images by a set of calls that let an image be read in by a scanner or displayed and have various parts of it saved, isolated, or transformed in various other ways.

Type faces and character sets: You control type faces in GDDM by using symbol sets. There are two types, vector symbol sets and image symbol sets. Vector sets are used only in graphics text. Image sets can be used in graphics and alphanumeric text.

GDDM supplies a number of symbol sets that you can use, and two symbol editors to produce symbols of your own. Here are the GDDM calls to use a Gothic symbol set, called ADMUUGEP, that is supplied by GDDM.

```
'GSCB 8 8'                      /* make character box large      */
'GSCM 3'                        /* set mode as vector text, type 3*/
'GSLSS 2 "ADMUUGEP" 193'        /* load symbol set, call it 193  */
'GSCS 193'                      /* say you are using set 193     */
message = "Dracula Lives"       /* set up the message            */
'GSCHAR 10 50 13 .message'      /* write the words               */
'ASREAD .typ .val .count'       /* send to terminal              */
```

Housekeeping and Query Calls: GDDM is completed by a set of housekeeping and query calls. The housekeeping calls allow control of how the device behaves at a detailed level. The query calls let called programs discover the current status. For example, the current position and the current set of attributes can be queried at the beginning of a subroutine and restored at the end, thus making the subroutine "safe" to use in any circumstances. Calls might be:

```
'GSQCP .oldx .oldy'             /* save current position         */
'GSQCOL .oldcol'                /* save current color            */
'GSQLW .oldwid'                 /* save current line width       */
```

and so on, for the color, line width, and other attributes.

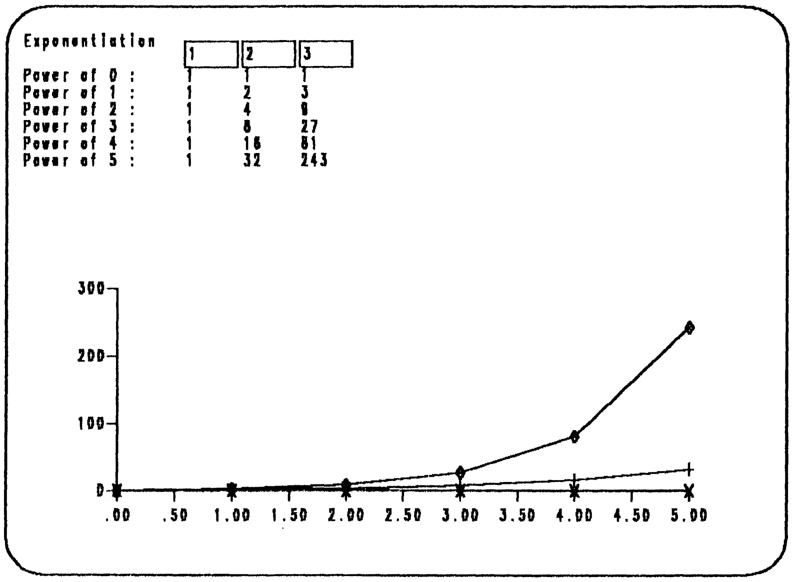
GKS interface: As well as the graphics interface shown above, GDDM also has an interface to GKS, the Graphical Kernel System. This is usable from GDDM-REXX in the same way as other calls. The calls use the FORTRAN binding names. Note, however, that GKS calls should not be mixed with GDDM graphics or charting calls. The GDDM-GKS product must be installed before GKS can be used.

Three-page GDDM-REXX

GDDM-REXX is an interface between GDDM and REXX. It allows you to freely mix GDDM calls and REXX instructions. It also contains subcommands to help with things like tracing and has a powerful syntax that simplifies the coding of complicated parameters.

Compared to other GDDM interfaces, GDDM-REXX is considerably easier to use because of a simplified REXX-like syntax, and because it is interpreted, making building of a program and debugging much less complicated.

The next page shows how the original sample in the REXX section (on page 5) can be updated to show full screen output, with values shown in a full screen table and also in a graph. Typical output is shown below, using a value of 3 for the argument. (Note that the argument should be restricted to values less than 10.)



```
Exponentiation   1     2     3
Power of 0 :     1     1     1
Power of 1 :     1     2     3
Power of 2 :     1     4     9
Power of 3 :     1     8     27
Power of 4 :     1     16    81
Power of 5 :     1     32    243
```

```
/* program to show exponentiation                                  */
Arg p                             /* get the argument p            */
Do j=1 to p                       /* for numbers up to p,          */
   Do i=0 to 5                    /*    for powers from 0 to 5     */
      k=i+1
      y.j.k=j**i                  /* put results in                */
                                  /*    multi-dimensional array y  */
   End i                          /* end inner loop                */
End j                             /* end outer loop                */

Address command 'GDDMREXX INIT'   /* see note 1                    */
Address gddm                      /* see note 2                    */
id=1
'ASDFLD .id  1  1  1  14  0'      /* see notes 3 and 4             */
'ASCPUT .id  .  "Exponentiation"' /* set up page heading           */

Do i=1 to p                       /* set up column headings        */
   id=id+1                        /* increment field id            */
   j=12+i*6                       /* j gives column number         */
   col=i//7                       /* remainder function to keep ... */
   If col=0 then                  /* color values between 1 and 7  */
      col=7
   'ASDFLD .id  2  .j  1  6  0'   /* set up fields for column heads */
   'ASFCOL .id  .col'             /* set colors                    */
   'ASFHLT .id  2'                /* reverse video                 */
   'ASCPUT .id  .  .i'            /* put numbers in headings       */
End i
                                  /* set up side column            */
Do i=0 to 5
   id=id+1
   j=3+i
   'ASDFLD .id  .j  1  1  15  0'  /* set up fields for left sides  */
   string='Power of' i ':'
   'ASCPUT .id  .  .string'       /* put words into side fields    */
End i
                                  /* now the values in the columns */
Do j=1 to p
   Do i=1 to 6
      id=id+1                     /* increment field id            */
      row=i+2
      col=12+j*6
      'ASDFLD .id  .row  .col  1  6  0' /* set up fields for values */
      'ASCPUT .id  .  .y.j.i'     /* put numbers into fields       */
   End i
End j
                                  /* now show it as a graph        */
'GSFLD 10  1  22  80'             /* set graphics field for chart  */
'CHPLOT .p  6  (0 1 2 3 4 5 6)  .y.' /* draw line chart - see note 5 */
'ASREAD .  .  .'                  /* send to terminal - see note 6 */
Address command 'GDDMREXX TERM'   /* see note 1                    */
Exit                              /* end of the program            */
```

Points to note when using GDDM-REXX

1. A **GDDMREXX INIT** command must be issued within the EXEC before GDDM is addressed and any GDDM calls are made. A **GDDMREXX TERM** command must be issued before any exit from the EXEC. Because **GDDMREXX** is a CMS command, it must be preceded by an **Address command** instruction unless the interpreter is already set up to send commands to CMS.

2. An **Address gddm** instruction must precede the first GDDM call.

3. GDDM calls should always be in quotes to avoid REXX making changes before the calls are passed to GDDM. The **GDDMREXX** command must be in quotes if you use any of the options with parentheses.

4. Variable names in GDDM calls and GDDM-REXX subcommands must be preceded by dots.

5. Array parameters can be passed in the following forms:

   - Element by element in parentheses. The elements may be either variable names or values. For example:

     ```
     'GSPLNE 3 (.left .xcenter .right) (10 .ycenter 20)'
     ```

   - As REXX stemmed variables, which are followed by a dot in the parameter string in the normal REXX manner. For example:

     ```
     'GSPLNE 3 .xarray. .yarray.'
     ```

     which is equivalent to:

     ```
     'GSPLNE 3 (.xarray.1 .xarray.2 .xarray.3) (.yarray.1 .yarray.2 .yarray.3)'
     ```

   - As names that GDDM-REXX will suffix with 1, 2, 3 and so on; there is more about this on page 56.

   Array parameters are more strictly interpreted in GDDM-REXX than they are in other programming languages. For example, arrays consisting of four sets of two values are treated as two-dimensional in GDDM-REXX although they can be treated as one-dimensional in other languages. If an array is passed by name, it will be indexed from element one irrespective of whether that is the first element, or whether it exists. More information is in the reference section (see page 56).

6. Some parameters can be replaced by dots, for example, lengths and counts that GDDM-REXX can calculate for itself, and returned values that a program does not refer to.

REXX-like tracing: The subcommand **'GXSET TRACE ON'** will start tracing. Tracing displays the REXX variable values that are received by GDDM-REXX, the values returned by GDDM-REXX, and the instruction being executed.

Special actions for mapping: Because GDDM-IMD does not define data structures in REXX, GDDM-REXX contains a utility that produces REXX-like data structures from GDDM-IMD. It also contains subcommands that can be used in the program to move values from REXX variables into the structure and vice versa. The utility is ERXMSVAR EXEC; the commands are **'GXSET MSADS'**, to move values into the structure, and **'GXSET MSVARS'**, to move values from the structure into variables.

Inside the back cover of this manual you will find a summary of the rules you need to follow when programming in GDDM-REXX.

part 1: learning

Learn by doing

The best way to learn about GDDM-REXX is by using it at a terminal.

Begin by running the samples of GDDM-REXX that are provided with the product. Then alter them to explore GDDM, REXX, and some of the special features of GDDM-REXX.

The following sessions take you through the basics of learning about GDDM-REXX. (Advanced programming considerations are discussed later.) The sessions are divided into two parts; the first part describes the background, and the second part tells you what to do.

Things to do are marked with a bullet, thus •.

If you have a terminal on which you can use GDDM-REXX, start using it now. If you have not yet installed GDDM-REXX, turn to "Part 2: Installation" on page 39.

If GDDM/VM has not been installed into a saved segment, you will need to issue a **GLOBAL TXTLIB ADMRLIB ADMGLIB** command for the GDDM libraries before you use GDDM-REXX. (Other GDDM programs may need other text libraries).

Session 1: Running the sample EXECs

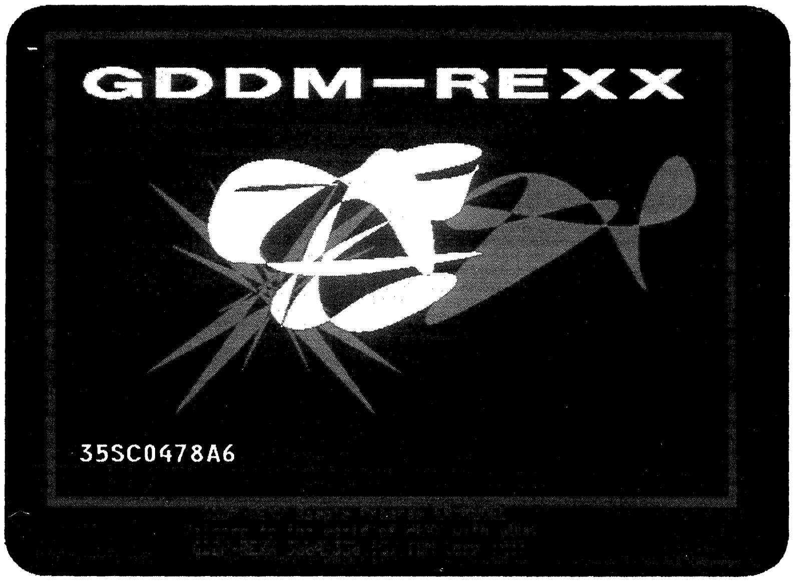
This session tells you how to run a GDDM-REXX EXEC and shows you some of the capabilities of GDDM-REXX.

The sample EXECs provided by GDDM-REXX are:

ERXMODEL	A model EXEC structure for users to pick up and change. Shows a GDDM-REXX picture.
ERXPROTO	An EXEC that defines the parameters of a specified GDDM call.
ERXTRY	An interactive EXEC that lets you try GDDM calls.
ERXMENU	A restaurant menu example using GDDM graphics and alphanumerics, from the *GDDM Application Programming Guide.*
ERXCHART	An example of calling the Interactive Chart Utility (ICU).
ERXOPWIN	A windowing example from the *GDDM Application Programming Guide.*
ERXORDER	An ordering example using mapping.

To run the EXECs, make sure you are linked to the GDDM-REXX disk and to the disks that contain GDDM. (If you do not understand what that involves, talk to your system programmer or other local VM expert.)

● Type in the name **ERXMODEL**. The result will look like the picture below. This is the model EXEC that you will copy and change as a way of exploring GDDM-REXX. (If the picture does not appear, see "If the session does not work" on page 16.)



● Try the other sample EXECs shown in the list above. The comments at the beginning of each EXEC tell more about them – you can type **ERXTRY ?** or **ERXTRY HELP** to see these comments.

● If you do not know GDDM, experiment with ERXTRY; it lets you type in GDDM calls and see what they do. Use the menu provided to get a list of calls available on your system. Some GDDM calls are listed in the description of GDDM in the previous chapter.

Note that ERXTRY normally has an open graphics segment in which it builds your experimental graphics programs. (A graphics segment is a group of graphics primitives that are handled together by GDDM; their use imposes restrictions on some GDDM calls.) As a result, if you use GDDM-PGF calls from ERXTRY, you must first issue a **GSSCLS** call. When you finish trying GDDM-PGF calls, issue **CHTERM** followed by **GSUWIN 0 100 0 100** to reestablish the GDDM window, and **GSSEG 0** to reopen the graphics segment. You cannot use GDDM-GKS calls with ERXTRY for similar reasons. ERXTRY can use a "log file" in which it places the calls you make; later you can use this log file with ERXTRY, or run it as an independent EXEC.

● The ERXPROTO EXEC will help you code GDDM-REXX; you supply the name of a GDDM call, and ERXPROTO shows the parameters in GDDM-REXX format. It can be used in CMS, in XEDIT, where it will add the call to the file you are editing, or in ERXTRY, where you type in the name of a call and then press PF11. A prototype statement is printed on the line. For example, if you type in **GSCHAR** you will get

```
'GSCHAR float float len3 char.len3'
```

You can change the values so that the statement reads:

```
'GSCHAR 50 60 12 "Good morning"'
```

50 is the X coordinate, 60 the Y coordinate, and 12 the number of characters in "Good morning" – more of the precise meaning of the syntax later in this chapter.

If the session does not work: First make sure you are linked to the correct disks, and that you have not made a typing error with the names. If you get error messages when running the EXECs, check whether there is a file called PROFILE ADMDEFS on any of the disks you have accessed. Rename it to something else if you can, or create a PROFILE ADMDEFS file on your A-disk that contains one blank line. (PROFILE ADMDEFS files control the format of GDDM output and can cause programs to give unexpected results in certain situations. For more information, refer to the *GDDM Guide for Users.*)

If GDDM/VM is not installed into a saved segment, you must issue a **GLOBAL TXTLIB ADMRLIB ADMGLIB** command for the GDDM libraries before you use GDDM-REXX; check with your system programmer.

Also, it is not a good idea to have an EXEC called GDDMREXX around – it can cause errors if it is interpreted wrongly.

Session 2: Building your first EXEC

This session lets you get a feel for GDDM-REXX EXECs by making small alterations to the ERXMODEL EXEC. Consult the listing of ERXMODEL on page 72 to see which areas of the EXEC can be changed.

● Copy ERXMODEL EXEC onto your own disk, and give it a name such as MODEL. Use a command of the form:

```
COPYFILE ERXMODEL EXEC fm MODEL EXEC A
```

where `fm` is the mode of the disk that contains GDDM-REXX.

Edit MODEL EXEC, using a command of the form:

```
XEDIT MODEL EXEC A
```

You will find the system editor XEDIT convenient for two reasons:

1. You can run the EXEC without leaving the editor, by first saving it and then typing its name on the command line. (You will need to specify `SET IMPCMSCP ON` in the editor.)

2. You can use ERXPROTO to add a model GDDM statement to the file you are editing, and then substitute values. For example, if you type in `ERXPROTO GSCOL`, `'GSCOL intg'` appears in your file, and you can substitute a color number for `intg`.

● Experiment with graphics.

Alter the `GSCOL` from `GSCOL 6` to `GSCOL 3`. This will make the lettering come out pink.

Comment out `Call mysubprog` and follow it with `GSLINE` calls to see the effect of drawing lines.

Draw a circle (which may be squashed, depending on the width-to-height ratio of your screen) with the calls:

```
'GSMOVE 30 40'                  /* move to point on circumference */
'GSARC 50 60 360'               /* arc centered at x=50, y=60,    */
                                /*    arc angle 360               */
```

Draw a shaded area by putting `'GSAREA 1'` before the drawing calls and `'GSENDA'` after them.

If you have any ADMGDF files on your disks, you can replace the model picture with another by changing the name `ERXMODEL` in the `GSLOAD` call to the name of another ADMGDF file, after reinstating the `Call mysubprog` statement.

Add the following code to your MODEL EXEC file, to make random pictures. Now MODEL will produce a different picture every time you run the program. The part to change is indicated in the listing on page 72.

```
string=time()
x = substr(string,1,2)
y = substr(string,4,2)
z = substr(string,7,2)
'GSCOL .x'
'GSAREA 1'
'GSPFLT . (.y .z .x .y .x ) (.x .y .z .z .y )'
'GSENDA'
```

Try other calls mentioned earlier in this book or in the Graphics Primitives section of the *GDDM Application Programming Guide.*

● Experiment with alphanumerics.

Find the **ASDFLD** and **ASCPUT** statements. These control the alphanumeric fields that say **Welcome to the world of....** Change the wording by altering the words in the **ASCPUT** statements, and change the location by altering the row positions in the **ASDFLD** statements. For example:

```
'ASCPUT 1 5 "Hello"'
```

(The **1** specifies field 1, and the **5** specifies the number of characters in field 1 – a dot may be coded in place of the 5.)

The fields in the sample allow input, because the last parameter in the **ASDFLD** call is **0** (unprotected). Therefore you can alter the contents by typing from your terminal and get the value into a REXX variable with an **ASCGET** call.

```
'ASREAD . . .'
'ASCGET 1 20 .inval'
```

You can then use the variable **inval** in REXX statements or GDDM calls. The REXX command **Say inval** types its value out on the screen.

You will find more about procedural alphanumerics in the *GDDM Application Programming Guide.* The sample program ERXMENU, which displays a restaurant menu, is a GDDM-REXX version of an example in that manual.

Session 3: Using the documentation

This session explains how to use the documentation for GDDM and for REXX, and for the CMS commands and facilities you will need to use with them.

References and how to use them: The table below shows the IBM manuals and other sources that will help you to code GDDM-REXX correctly.

Reference source	Notes
GDDM	
GDDM Base Programming Reference	The definitive reference source for GDDM/VM calls.
GDDM Application Programming Guide	Essential to put calls in context, showing examples of the calls in use.
GDDM Base Programming Reference Summary	A summary of the base calls; contains data on call syntax, including parameter types.
GDDM-PGF Programming Reference	The definitive reference source for GDDM-PGF calls.
GDDM-PGF Programming Reference Summary	A summary of the GDDM-PGF calls; contains data on parameters.
GDDM-GKS Programming Guide and Reference	The definitive reference source for GDDM-GKS calls.
Online information for GDDM	ERXTRY EXEC lets you try GDDM calls and see the results (you can also use REXX statements). ERXPROTO EXEC returns the parameters of GDDM calls as GDDM-REXX statements in the correct format; it can be used from CMS, ERXTRY, or XEDIT.
REXX	
VM/System Product Interpreter User's Guide	Guidance on learning to use REXX.
VM/System Product Interpreter Reference	Definitive reference information for REXX.
Online information for REXX	**HELP REXX command** supplies help with REXX commands; **HELP REXX MENU** gives more general help.
CMS	
Online information for CMS	HELP is available for all CMS commands. (You will need these commands to manage files and to read data from them.)

Other manuals which may be of interest are listed at the front of this book.

Reference manuals and GDDM call syntax: The GDDM programming reference manuals use an abstract syntax to describe the calls, because they are intended for programmers using GDDM with various languages. When referring to the manuals, keep in mind that:

1. Parameters are shown in parentheses, with commas between them. Neither the commas nor the parentheses are used in GDDM-REXX.
2. The descriptions of the parameters have more information about data types than is required by GDDM-REXX (for example, that they are "*short* floating point").
3. Array parameters are not described as rigorously as GDDM-REXX requires. GDDM-REXX pays strict attention to the number of dimensions or **rank** of arrays.

Here is an example of typical information from the reference manuals and how to interpret it for GDDM-REXX.

```
GSPLNE (count,xarray,yarray)
```

count *(specified by user) (fullword integer)*
... explanation

xarray *(specified by user) (array of short floating point numbers)*
yarray *(specified by user) (array of short floating point numbers)*
... explanation

Equivalent information is given by the ERXPROTO EXEC which gives the output:

```
'GSPLNE cntl float.cntl float.cntl'
```

There are three parameters in each case, and they are equivalent. **Count** or `cntl` means the number of items in an array. In the reference manual, *integer* means that the parameter must be a whole number, and *specified by user* means that it is a value that must be passed to GDDM. `float.cntl float.cntl` in ERXPROTO is equivalent to the reference manual's **xarray yarray**. The dots after `float` show that they are arrays; the fact that only one `cnt` follows the dot shows that the array has one dimension. The word `float` (equivalent to *short floating point* in the reference manual) means that any number can be used; GDDM-REXX does not require a whole number.

In the reference manuals the phrases *specified by user* and *returned by GDDM* are used to show whether it is a value that is passed to GDDM, or one that is generated by GDDM and returned. In ERXPROTO, values that are returned by GDDM are preceded by a dot, for example `'ASREAD .intg .intg .intg'`; `ASREAD` gives information on how the user returns control to the program (which PF key was pressed, for example). For returned values, you can pass a variable name if you are interested in the value, or a dot if you are not.

Examples in Application Programming Guide: The *GDDM Application Programming Guide* contains example programs which are the normal starting point for learning about a particular aspect of GDDM. Many of these examples are in PL/I. To transform these into GDDM-REXX, you should:

1. Remove the parentheses and commas from the GDDM calls, and the semicolons if there is only one statement per line.
2. Remove the DCL statements, substituting assignment statements where the DCL contains INIT.
3. Remove the `FSINIT` and `FSTERM` calls. You will need to supply `GDDMREXX INIT` and `TERM` commands.
4. Remove the %INCLUDE statements and other statements that are PL/I-only, taking care with PL/I labels and array handling.

Here is an example:

```
PL/I                              GDDM-REXX equivalent

A:PROC OPTIONS (MAIN);            /* REXX comment              */
CALL FSINIT;                      Address command 'GDDMREXX INIT'
                                  Address gddm
DCL X,Y;                          /* remove simple DCL         */
DCL Z INIT (3);                   z=3
X=10; Y=10;                       x=10; y=10
CALL GSLINE(X,Y);                 'GSLINE .x .y'
CALL GSLINE(20,Z);                'GSLINE 20 .z'
%INCLUDE ADMUPINA;                /* remove                    */
CALL FSTERM;                      Address command 'GDDMREXX TERM'
```

● Compare the listing of a number of calls in the *GDDM Base Programming Reference* with those given by the sample ERXPROTO EXEC. If you use the ERXTRY EXEC you can type in the call name and press PF11 for the syntax, or you can use the menu.

● Compare the sample ERXMENU EXEC with the same program in the *GDDM Application Programming Guide.* Look in the index under "alphanumerics, procedural," as a secondary heading "menu example." These are essentially the same programs, one in GDDM-REXX and the other in PL/I.

● Try converting some or all of a PL/I example in the *GDDM Application Programming Guide* to GDDM-REXX. Note that if you want a REXX expression to be evaluated, you must code it outside the quotes that normally surround a GDDM call. For example:

```
'GSLINE'  x-5.0  y+5.0
```

Session 4: Messages and debugging

This session tells you how to recognize and avoid errors. It also describes how to use REXX and GDDM-REXX tracing to detect bugs that are not obvious.

Messages: When you are using GDDM-REXX you can get four types of message.

`ERX...`	These come from GDDM-REXX itself; they are caused by things such as giving the wrong number of parameters, or making a typing error in a GDDM call. They are listed on page 70.
`+++Error...`	These come from REXX and are caused by things such as getting quotes in the wrong place, or operating on uninitialized variables.
`ADM...`	These come from GDDM and are normally caused by passing incorrect values to GDDM. See the *GDDM Messages* manual.
`DMS...`	These are caused either by leaving the comment off the top so that the EXEC is not interpreted by REXX, or by CMS commands issued in REXX.

Note that some messages are affected by the CP commands `SET EMSG` and `SET IMSG`.

Quotes: Getting quotes in the wrong place or omitting them is a common cause of error with GDDM-REXX. We suggest that you always put GDDM calls in single quotes, and put double quotes around character strings within the calls. GDDM calls will frequently work without quotes, but leaving them out can lead to errors. A particular trap is passing negative numbers which are taken by REXX to be subtraction operations. Another possible source of error is using the name of a GDDM call as a REXX variable.

Dots and omitting them: Getting dots in the wrong place is another common error. REXX does not allow dots before variable names, except in call statements.

In GDDM calls, leaving out a dot before a variable name will give an error message, *if the call is enclosed in quotes*. If the call is not in quotes, the value of the REXX variable will be passed (which is probably what you want).

Another source of errors is putting a dot before a numeric value in a GDDM call. `'GSLINE .500 .y'` for example, is an easy mistake to make if you are changing a variable like `.x` for a value, say five hundred. `.500` is then treated as one-half and the results may surprise you.

Disappearing messages: GDDM clears the screen when it opens the device. This may prevent you reading error messages caused by previous GDDM calls or REXX statements. To avoid this for GDDM error messages, place an `FSFRCE` call immediately after `Address gddm`. (This call should be removed when the program has been debugged.) For REXX error messages use a statement that will request input from the terminal such as `Parse pull dummy`.

Tracing: Tracing is available in both REXX and GDDM-REXX. You can use either or both. Tracing is also available within GDDM itself; this is described in the *GDDM Diagnosis and Problem Determination Guide*; however, REXX and GDDM-REXX tracing should suffice for normal program debugging.

Here is an example of how to turn on REXX tracing for interactive tracing of results:

```
Trace ?r
x = x + y
'GSLINE' x y
```

To stop tracing type `Trace off` at any stage. You can also enable REXX tracing by the command `SET EXECTRAC ON` – see the *VM/System Product Interpreter Reference* manual for details.

GDDM-REXX tracing is started with the command

```
'GXSET TRACE ON'
```

and ended with

```
'GXSET TRACE OFF'
```

Always use the quotes to be safe. Here is an example of some typical output:

```
Statement                    Resulting trace output

'FSQDEV . (.a .b .c .d)'     ERX0000 I Var set: a = "1"
                             ERX0000 I Var set: b = "2"
                             ERX0000 I Var set: c = "32"
                             ERX0000 I Var set: d = "80"
                             ERX0000 I "FSQDEV . (.a .b .c .d)"

s='A STRING variable'        ERX0000 I Var fetch: s = "A STRING variable"
'gschap . .s'                ERX0000 I "gschap . .s"
```

There is more detailed information about tracing in "Part 3: Diagnosis" on page 47.

● Make deliberate mistakes to get an understanding of what happens. Create a very simple EXEC of the form:

```
/* error test exec                                                    */
Address command 'GDDMREXX INIT'
Address gddm
'FSFRCE'
'GSCOL 6'
'GSLINE 20 20'
'ASREAD .typ .val .count'
Address command 'GDDMREXX TERM'
Exit
```

1. Remove a quote from one end of a GDDM call. You will get a REXX message either about unmatched quotes, or about a phrase being too long. Note that the line number of the trouble is given at the start of the message.
2. Add an extra parameter to a `GSLINE` call. You will get a GDDM-REXX message (starting "ERX") listing the call and giving the error.
3. Insert a GDDM call with incorrect parameters such as `'GSCOL -500'`. This will give you an error message from GDDM, which begins with "ADM".
4. Add a blank or other line at the start of the EXEC. The EXEC is then processed by the CMS EXEC handler and will give a series of CMS messages (typically `INVALID CHARACTER IN MODULE NAME`.)

If you try these changes on the model EXEC (ERXMODEL), you will find that the messages give the line number where the error occurred. This is because of the following code, which you may want to include in your own programs:

```
/* at start                                                    */
Signal on error                  /* intercept non-zero return  */
                                 /*    codes from GDDM-REXX    */
   .
   .
error:                           /* after exit of mainline exec  */
grc=rc
'GXGET LASTMSG .g_msg'
Say 'Line:' sigl '-' sourceline(sigl)
Say 'Non-zero return code from GDDM-REXX call: ' grc
Say g_msg
Address command 'GDDMREXX TERM'  /* Terminate GDDM-REXX        */
Exit 99
```

● Try trace using the ERXMODEL EXEC.

You can start interactive REXX tracing of results with the statement:

```
Trace ?r
```

Place it at the start of the EXEC.

Start and end GDDM-REXX tracing thus:

```
'GXSET TRACE ON'                 /* do NOT omit the quotes       */
                                 /* must come after GDDMREXX INIT */
                                 /* and ADDRESS GDDM             */
   .
   .
'GXSET TRACE OFF'                /* do NOT omit the quotes       */
```

Place the GXSET subcommand after GDDM-REXX is initialized and GDDM is addressed; otherwise it will not be recognized.

Session 5: Trying out GDDM-REXX parameters

This session will help you understand GDDM-REXX parameter handling so that you can code your EXECs efficiently. You will need to refer to the description of parameter syntax in "GDDM call syntax" on page 54.

The advantages of GDDM-REXX parameters: There are three main things about GDDM-REXX parameters that make them easy to use. They are:

1. Not having to supply lengths and counts for strings and arrays of values passed to GDDM − you can use dots instead.
2. Being able to use dots for returned values you do not care about.
3. Being able to handle array parameters by listing them element by element in the call.

Potential problems with GDDM-REXX parameters: GDDM-REXX parameters can cause problems because of the power and complexity of some GDDM calls. If you code parameters incorrectly, particularly array parameters, GDDM-REXX will normally give you an error message. Sometimes, however, it may interpret them in unexpected ways, without giving an error message.

Here are the most likely messages caused by getting a parameter wrong in a GDDM-REXX call.

```
ERX0002 E Too few parameters
ERX0003 E Too many parameters: '........'
ERX0004 E Invalid parameter type: '........'
ERX0005 E Invalid integer value: '........'
ERX0006 E Invalid real value: '........'
ERX0008 E Parameter rank too large: '........'
ERX0009 E Invalid parameter syntax: '........'
ERX0011 W REXX variable had no value: '........'
ERX0013 E Variable name required for return parameter: '........'
ERX0018 E Right parenthesis missing
ERX0020 E Ending string delimiter missing
```

If you get unexpected results, try tracing. Put `'GXSET TRACE ON'` before the call that is giving trouble, and `'GXSET TRACE OFF'` after it.

The trace will show the values GDDM-REXX got, and the values it sent on to GDDM. You can use ERXPROTO to see what parameters are required by a particular call.

● To experiment with omitting parameters, use the interactive ERXTRY EXEC. These examples will work:

```
'GSCHAR 50 50 . "Hello"'

'ASDFLD 1 10 17 1 20 2'
'ASCPUT 1 . "Hello"'
```

If you omit the length from `ASCGET`, you will get an error message. GDDM requires that lengths of returned values must be specified, so they cannot be omitted for returned values. So this is wrong:

```
'ASCGET 1 . .var'
```

You can experiment with omitting returned values in the same way. From ERXTRY, try the call

```
'FSQDEV 4 ( . . .r .c)'
```

and then the REXX statement

```
Say 'rows on screen are' r 'columns on screen are' c
```

Handling array parameters: To find out about handling arrays, modify a copy of the model EXEC (ERXMODEL). The important points are:

1. Using stemmed variables:

   Try experimenting with any call that takes a one-dimensional array.

   ```
   procopts.1=1000
   procopts.2=2
   'DSOPEN 9 1 * 2 .procopts. . ( )'
   ```

   Then try further calls with two-dimensional arrays (see also paragraph 4 on page 27).

   ```
   'ASQFLD 2 3 4 .array.'            /* sets values in array.1.1      */
                                     /* through array.3.4             */
   ```

2. Enumerating array values within the call:

   A good call to use for coding directly into the parameter string is GSPLNE, which draws a series of straight lines. Try:

   ```
   x1=10
   /*      cntl   float.cntl    float.cntl      - syntax from ERXPROTO     */
   'GSPLNE 3     (.x1 20 30)   (40 50 60)'
   ```

   If you specify an explicit count value for an array, the array is truncated or expanded with zeroes to match. For example:

   ```
   'GSPLNE 3     (5 6 7 8)     (6)'
   ```

   is passed to GDDM as

   ```
   'GSPLNE 3     (5 6 7)       (6 0 0)'
   ```

3. Replacing dimension information with dots:

   Because GDDM-REXX can count the numbers in the parentheses, you can replace the count with a dot:

   ```
   'GSPLNE .     (.x1 20 30)   (40 50 60)'
   ```

4. Defining stemmed variables for columns of two-dimensional arrays:

   Some GDDM calls require parameters in the form of a two-dimensional array. For example, **ASDFMT** allows a number of alphanumeric fields to be defined; the parameters are a count of the number of fields, a count of the number of elements being defined for each field, and an array that contains the element values. This call defines two fields with five elements provided for each field:

   ```
   'ASDFMT 2 5 ((1 1 1 80 1) (2 2 1 80 1))'
   ```

   Many programs need to create a number of similar alphanumeric fields, one below another. You can use single values for columns in multi-dimensional arrays, as in this example of **ASDFMT**, to set up ten fields numbered 1 to 10 at the top of the screen:

   ```
   Do i= 1 to 10                        /* set up array with values 1-10  */
      nums.i=i
   End i
   /*    fields  count    ids     row   col depth width  type             */
   'ASDFMT 10      6     (.nums. .nums.  1  1      80      1  )'
   /* Fields 1 to 10 are set up in rows 1 to 10 respectively.  Each       */
   /* field starts in column 1, is one row deep, eighty characters wide */
   /* and has the field type of 1 (alphanumeric output, numeric input)  */
   ```

Array parameters are discussed more fully in the reference section of this book.

More advanced programming

This section describes how to use two aspects of GDDM-REXX that differ significantly from the usage of GDDM with other languages:

1. Mapping using GDDM-IMD
2. Instances and reentrancy.

Mapping

This section assumes that you are familiar with GDDM Interactive Map Definition, and that you want to use maps in your EXECs. There is more information on mapping in the *GDDM Application Programming Guide*. You can learn about GDDM-IMD in *GDDM Interactive Map Definition* and in the online tutorial that is supplied with the product.

Because REXX does not have data structures, special provisions have to be made for mapping in GDDM-REXX. This consists of two subcommands and an EXEC:

GXSET MSADS	Subcommand to move data from REXX variables into a structure that can be used for mapping.
GXSET MSVARS	Subcommand to move data back into the REXX variables.
ERXMSVAR	EXEC to create a set of named REXX variables that can be included in a mapping EXEC.

Producing a GDDM-REXX mapping program involves the following steps. The illustrations are taken from GDDM-IMD panels or from the sample mapping ERXORDER EXEC.

1. In GDDM-IMD, create a mapgroup (called **mapgrpn**) and a map (**mapn**). Generate the mapgroup. Your generated mapgroup must contain field names; you can specify any of the programming languages that IMD supports.

   ```
   FIELD NAMES IN GENERATED MAPGROUP ==> YES
   ```

2. Use the ERXMSVAR EXEC (see page 66) to create a REXX data structure:

   ```
   ERXMSVAR mapgrpn mapn X_
   ```

   X_ was used in ERXORDER. It is used as a prefix in constructing variable names; any suitable prefix can be used.

3. Copy the REXX data structure into your EXEC. The file containing the structure will have the filename of your mapgroup and the filetype of GDDMCOPY. Note that the length of the application data structure, needed by some mapping calls, is included in the file.

```
/* GDDM-REXX: output from ERXMSVAR EXEC ....                     */
/* .... MAPGROUP: ERXORDD6, MAPNAME: ERXORDER                    */
X_=""
X_PROD.1 = "           "
X_DESC.1 = "                     "
X_COST.1 = "           "
X_QTY.1 = "     "
X_DESC.2 = "                     "
   ..
X_ASLENGTH=429                    /* length of ADS string        */
```

4. Change the values in the application data structure, if you want initial values in the map. They are set to blank by ERXMSVAR. Set up mapping page and fields in GDDM:

```
X_desc.1='NUT, HEX, 4-40  '       /* set some initial values     */
X_desc.2='BOLT,FLATHEAD, 4-40      '
'MSPCRT 3 -1 -1 "ERXORDD6"'       /* create mapped page - size   */
                                  /* of map taken from mapgroup  */
'MSDFLD 9  1  1 "ERXORDER"'       /* create a mapped field       */
```

5. Use **GXSET MSADS** to collect the data into a variable that can be used in the mapping calls. GXSET MSADS takes the values in the REXX data structure and puts them in a character-string variable that will be acceptable to GDDM mapping calls:

```
'GXSET MSADS ERXORDD6 ERXORDER .X_ .ads'
```

6. Use GDDM mapping calls to display the map and read the input:

```
'MSPUT 9 0 . .ads'
'ASREAD .at .am .mf'
'MSGET 9 0 .X_aslength .ads'
```

7. Use **GXSET MSVARS** to move the data from the mapping call variable into the REXX variables in the data structure where you can work on them:

```
'GXSET MSVARS ERXORDD6 ERXORDER .X_ .ads'
If X_qty.1 ¬= '    ' then ...     /* handle variables            */
```

Alternative method: The method just shown is the simplest and provides you with a list of the names you used in the map in your EXEC, but you can do without the ERXMSVAR EXEC as shown in the sample that follows:

```
/* alternative way to initialize the REXX data structure           */

ads=''                               /* just define ads variable, so   */
                                     /* it will not produce message:   */
                                     /* ERX0011 W REXX variable had    */
                                     /*    no value: 'ads'             */

'GXSET MSVARS .mygrp .mymap .X_ .ads' /* initialize all variables in */
                                     /* map to blanks                  */

X_prod.1='99999999999'               /* set individual values, as      */
                                     /* required (optional)            */

'GXSET MSADS  .mygrp .mymap .X_ .ads' /* then, initialize the ads     */
X_aslength=length(ads)               /* remember length for MSGET      */

/* set up mapped field and other items as previous example           */

'MSPUT 9  0 . .ads'
'ASREAD .at .am .mf'
'MSGET 9 0 .X_aslength .ads'
'GXSET MSVARS .mygrp .mymap .X_ .ads' /* get values placed in REXX    */
                                     /* variables                      */
```

This method requires one less step to produce your mapping program but has the disadvantage that there is no record of the variable names within your program.

Things to avoid: Mapping needs some care, particularly if you are new to it. Take care over the following aspects:

1. The name of your mapgroup – it may not be what you expect. GDDM-IMD applies a device suffix to the name you first use. You can tell the mapgroup when you list your files by the fact that it has a filetype of ADMGGMAP.
2. The mapgroup and map names are GDDM tokens, and must be passed to GDDM in uppercase. Ensure that you do this.
3. The GDDM application data structure (called `ads` in the example) is set by the `GXSET MSADS` subcommand. Do not alter it by REXX assignment statements.
4. The prefix used in the `GXSET MSADS` and `GXSET MSVARS` subcommands must be that used when running the ERXMSVAR EXEC. Avoid using dots at the end of the prefix. This will make REXX treat the items in the map as stemmed variables, with potential substitution taking place. `X_` is a safer prefix than `X.` (with a dot).

Experimenting with the samples: GDDM-REXX has three sample items that are related to mapping. They are:

ERXORDER EXEC	The sample EXEC.
ERXORDD6 ADMGGMAP	The mapgroup used by the EXEC. D6 is the suffix for 32-row by 80-column displays.
ERXORDER ADMIFMT	The import MSL (map specification library in portable form).

To experiment with the map, you have to use GDDM-IMD to IMPORT it. Then you can edit and change the map and the sample EXEC to experiment with mapping.

1. Start GDDM-IMD using a new MSL name: `ADMIMD anyname`
2. Press ENTER until you get to the directory panel 0.1. Type `I` in the commands column on the left (nothing else). Press ENTER for the IMPORT panel where you fill in the name `ERXORDER`. This will import the map.
3. Before you change it, you should rename it. Otherwise the sample will not work properly if you generate a changed mapgroup. Use `R` against the group in the directory panel and choose a suitable name in the new panel.

When you have done this, you can experiment with the new mapgroup and a copy of the sample ERXORDER EXEC.

Multiple instances of GDDM and GDDM-REXX

You can have multiple instances of GDDM and of GDDM-REXX. This facility can be used to create programs that run a number of independent applications, each with its own environment.

Multiple instances of GDDM and GDDM-REXX can be controlled separately. You can have multiple instances of GDDM within one instance of GDDM-REXX. You can also have multiple instances of GDDM-REXX.

Instances of GDDM: Instances of GDDM are controlled by GDDM-REXX, using the reentrant interface to GDDM. GDDM-REXX allows simplified access to this by the `GXGET AAB` and `GXSET AAB` subcommands.

An application can jump between instances by using a `GXSET AAB` subcommand with the application anchor block (AAB) of the instance. Instances are chained together such that the default instance is always the first in the chain (and cannot be terminated by `FSTERM`). When a new instance is created, it is added to the end of the chain and becomes the current instance. If an instance is terminated by an `FSTERM` call, the chain is remade, and control returns to the previous instance in the chain.

Here is a simple EXEC that demonstrates GDDM instances and lets you move between them by using the PF keys. The variable v in the ASREAD calls is returned with the number of a PF key. If you press anything other than PF1, 2 or 3, you will leave the EXEC.

```
/* REXX EXEC to demonstrate multiple instances of GDDM              */
Address command 'GDDMREXX INIT'
Address gddm
Do i = 1 to 3                      /* set up 3 instances of GDDM    */
   'GSCOL .i'                      /* draw a colored line           */
   'GSLINE 50 50'
   'GXGET AAB .name.i'             /* extract the AAB               */
   'FSINIT'                        /* start another instance        */
End i
'GSCHAR 15 50 . "PF1 2 or 3 to select GDDM instance 1, 2, or 3"'
'GSCHAR 15 40 . "Any other PF key to end"'
'ASREAD .a .v .'                   /* V gets number of PF key       */
Do forever
   If a¬=1|v>3 then                /* leave if not PF 1, 2, or 3    */
      Leave
   'GXSET AAB .name.v'             /* select an instance            */
   string= 'this one is' v
   'GSCHAR 40 50 . .string'
   'ASREAD . .v .'
End                                /* do forever                    */
Address command 'GDDMREXX TERM'
Exit
```

Instances of GDDM-REXX: Instances of GDDM-REXX are controlled by the GDDMREXX command. GDDMREXX INIT starts a new instance; GDDMREXX TERM terminates the latest instance. It is not possible to jump between instances. Only the latest instance can be used, and when that is terminated the previous one will be activated – they are on a push-down stack.

You can experiment with instances of GDDM-REXX using the EXEC shown above. Simply place an outer loop around the whole EXEC. Notice, however, that you must use an Address command statement before you use a GDDMREXX command.

Termination: Proper termination of instances of both GDDM and GDDM-REXX is important. All instances of GDDM that are active are properly terminated by a GDDMREXX TERM command. However, it is important that each instance of GDDM-REXX is terminated at any possible exit including error and abnormal exits. Failure to terminate GDDM-REXX can result in storage being used up progressively. GDDM-REXX is automatically terminated on return to CMS when you get the ready message; however, if you are working from FILELIST or a similar program that does not return to CMS command ready, it is not terminated. If you suspect that you have multiple instances of GDDM-REXX around, you can terminate them all with the GDDMREXX TERM (ALL) command, or check using the NUCXMAP command for entries starting with ERX, possibly with a preceding blank.

Hints on using GDDM-REXX

Producing reliable, safe, independent EXECs

If you are producing EXECs which are to be used widely and thus could be invoked from either the CMS command line or from within other EXECs, take care with the initialization and termination of GDDM-REXX. Some guidelines are:

1. Ensure that matching `GDDMREXX INIT` and `GDDMREXX TERM` commands are issued along every possible path through an EXEC. This includes terminating GDDM-REXX within any error exit. A method of trapping errors within EXECs is shown in the listing of ERXMODEL at the back of this book. Failure to properly terminate GDDM-REXX may cause a calling EXEC (which was using its own instance of GDDM-REXX) to continue its task using the incorrect instance.

2. Always use an **Address gddm** statement prior to any GDDM call or GDDM-REXX subcommand. The default environment for REXX EXECs is CMS, so failure to issue **Address gddm** will cause the calls to be misdirected (probably resulting in a return code of −3).

3. Always place GDDM call statements within quotes. This prevents any possible substitution of values by REXX.

The model EXEC (ERXMODEL) contains a set of recommended statements for use in initializing (prolog code) and for terminating (epilog code). If you use these, you will meet all these guidelines without having to think about them. In addition, ERXMODEL contains a suggested method of intercepting errors. A listing of ERXMODEL is given on page 72.

For your own use you may prefer to have shorter prologs and epilogs than those used in the sample EXECs. Suggestions for short versions are shown below. They are not recommended for EXECs that are to be widely used.

```
Address command 'GDDMREXX INIT';Address gddm /* simple prolog - see      */
                                             /* ERXMODEL for full prolog */
Address command 'GDDMREXX TERM';Exit rc      /* simple epilog            */
```

You can find out whether GDDM-REXX is already active by using the `SUBCOM` command. If it is, you might want to use the existing instance, rather than initializing your own instance, with the overheads involved. The ERXPROTO EXEC contains such a test. Try

```
Address command 'SUBCOM GDDM'
If rc <> 0 then ...              /* GDDM not initialized, load it     */
```

How to tackle a programming task

If you have a particular task to do using GDDM-REXX you should study the *GDDM Application Programming Guide* and look at the GDDM-REXX samples for anything that is similar. This book provides only a very brief introduction to the facilities of GDDM.

Controlling access to subset mode

GDDM-REXX EXECs should be coded in a way that prevents the user getting into CMS subset mode by pressing PA2 (the default method of getting into subset mode for GDDM). This is because further GDDM-REXX EXECs cannot be used in subset mode when a GDDM-REXX EXEC is suspended. The attempt is diagnosed and an error message is given.

The **DSOPEN** call in the sample ERXMODEL EXEC will prevent access to CMS subset from GDDM. So will an entry like this in your PROFILE ADMDEFS file:

```
          DEFAULT PROCOPT=((CMSINTRP,NONE))
```

It is then possible to test for the use of PA2 after every read, and then go into CMS subset in a controlled manner with the calls:

```
'ASREAD .typ .val .count'
If typ = 4 & val = 2 then            /* typ=4 means PA key,        */
                                     /* val=2 is key 2             */
   Address command 'SUBSET'
```

Using GDDM-REXX from CMS subset or other programs

If you intend to use GDDM-REXX EXECs in subset mode, or by calling them from other programs, you should first load the GDDMREXX command module in the nucleus. This can be done with the command **NUCXLOAD GDDMREXX**. The module is not large and the command can be included in a PROFILE EXEC. The reason you may need to do this is that the first time you use GDDM-REXX, it loads the module at address X'20000' unless that module is already in the nucleus. (GDDMREXX then loads itself into the nucleus for subsequent use.) This may interfere with the use of that part of virtual storage by other programs. When you have finished using your GDDM-REXX EXECs, you can issue the **NUCXDROP** command to free nucleus storage.

Prototyping or production applications

GDDM-REXX is interpreted, and this means that performance is not as good as that of efficient compiled or assembled code. However, speed of producing a solution is often as important as fast execution, and in this respect GDDM-REXX should be better than other methods of writing applications which use GDDM. Its neat syntax and the power of the REXX language make it perfectly suitable for many applications.

If you are using GDDM-REXX for prototyping you should bear in mind the discussion of coding styles below.

Coding styles — strict or loose syntax

With GDDM-REXX you have a choice of coding styles. If you are coding your own private EXECs you can minimize the time spent in typing by replacing parameters with dots and putting array and other values directly into the GDDM calls. In fact you can go further, and provided you understand REXX's rules, you can sometimes leave out the quotes around GDDM calls. However, be aware of the potential difficulties when doing this; REXX may attempt substitution of values in place of names, and negative values may cause REXX to attempt subtraction.

If you are producing prototype code that will later be recoded in another language, all parameters should be passed by name:

```
count=3
xarray.1=20; xarray.2=30; xarray.3=40;
yarray.1=40; yarray.2=50; xarray.3=60;
'GSPLNE .count .xarray. .yarray.'
```

and not:

```
'GSPLNE . (20 30 40) (40 50 60)'
```

This will minimize the difficulties of conversion. The main remaining problem will be producing the data declarations for the target language. See page 21 for guidance on PL/I equivalents.

Common errors

Parameters incorrectly passed: There are several causes for this. Common ones are:

1. Passing arrays with the wrong number of dimensions — for example, passing a scalar value instead of an array or a one-dimensional array instead of a two-dimensional array. Arrays which have only one member, or two-dimensional arrays with only one member in one dimension, frequently cause errors. More about this in "GDDM call syntax" on page 54.
2. Failure to pass character strings in uppercase. GDDM requires many strings such as symbol set names, mapgroup names, options for CHSET and other calls to be in uppercase.
3. Incorrect typing of quotes or dots. All GDDM calls should be included in quotes.
4. Prefix minus signs outside quotes. These can be interpreted as infix minuses, so either use quotes, or code `-32` as `0-32`.

Storage is used up: When GDDM-REXX EXECs are run from a program such as FILELIST, the instances of GDDM are not cleared if the EXECs do not issue an `Address command 'GDDMREXX TERM'` command at the end of execution; include the `(ALL)` option if you think this is happening. More about this in "Multiple instances of GDDM and GDDM-REXX" on page 31.

Unexpected messages: Apparently meaningless messages from GDDM can be caused by having a PROFILE ADMDEFS file on one of your disks that causes GDDM to send output for the wrong type of device. See also page 16.

Unexpected pictures: If you use the `hi` (halt interpretation) immediate command, you should issue a `GDDMREXX TERM` command. Otherwise, you may occasionally find that a picture you had been looking at some time previously reappears on the screen, or a new picture, nothing to do with your current activity, appears. This is because GDDM programs that were suspended become active again and send "old" pictures to the screen.

Numerical data not being recognized: By default, GDDM pads strings with nulls (character X'00') when the full length has not been keyed in at the terminal. This stops REXX recognizing them as numbers, and causes errors. The simplest solution is to use the **ASDFLT** call before any of the fields are defined. One of its parameters allows the nulls to be translated to blanks, which are acceptable to REXX. The code required is:

```
default_array. = -1             /* leave other defaults alone (-1)*/
default_array.8 = 1             /* set parameter 8 nulls to blank */
'ASDFLT 8 .default_array.'      /* make conversion the default    */
'ASDFLD etc etc'                /* must follow                    */
```

An alternative would be to use the REXX **STRIP** function to remove the nulls, or the GDDM **ASFIN** call to convert nulls to blanks.

Difficult calls:

ASDFMT (which defines alphanumeric fields) needs care because its third parameter is a two-dimensional array. There is a temptation to code it as a one-dimensional array if only one field is being defined. See the examples in "Parameters that are too short" on page 57 for further details.

ASGGET (which gets the contents of double-byte character string fields) needs care because the length parameter is the number of DBCS characters, so twice that number of bytes are returned.

ASQFLD (which queries the attributes of a field) needs care because the dimensions of the array it passes depend on the first argument which is called the **code**. Examples of the different types of code and calls for them are given below:

```
/* CODE 0                                                      */
/* query one field specifying id of field in second argument   */
/* array is one dimensional and returned by GDDM               */
'ASQFLD 0 1      3      .onedim.'

/* CODE 1                                                      */
/* query several fields listing ids in column of array - the array  */
/* must first be initialized as it is partially specified by user   */
twodim.=0                       /* initialize array before sending*/
twodim.1.1=1                    /* first field to be queried is 1 */
twodim.2.1=2                    /* second field to be queried is 2*/
'ASQFLD 1 2      3      .twodim.'

/* CODE 3 and 4                                                */
/* query all fields and have number and values returned in array    */
/* array need not be initialized. it is totally returned by GDDM    */
'ASQFLD 3 2      3      .twodim.'
```

CHTOWR (which draws tower charts) needs a three-dimensional array, the first dimension of which is one. Here are an example of the syntax from ERXPROTO and examples of the call:

```
'CHTOWR cnt1 cnt2 cnt3 float.cnt2  float.cnt3  float.cnt1.cnt2.cnt3'
'CHTOWR 1    2    3    (4 5)       (6 7 8)     (((9 10 11) (12 13 14)))'
'CHTOWR 1    2    3    .onedima.   .onedimb.   .threedim.'
```

CHXLAB (which lets you specify labels for the X axis on charts) can cause problems to users used to other languages (similarly CHYLAB, CHXDLB, CHZDLB, CHKEY, CSCHA, and CSQCHA). In GDDM-REXX, the last parameter is an array of strings of the length specified. To use a single string containing all the values is an error. Here are an example of the syntax from ERXPROTO and an example of the call:

```
'CHXLAB cntl len2    char.cntl.len2'
'CHXLAB 6    3       ("JAN" "FEB" "MAR" "APR" "MAY" "JUN")'
```

GSVECM (which draws a series of vectors) has a first argument that specifies the number of rows in a two-dimensional array that always has three columns. Here are an example of the syntax from ERXPROTO and examples of the call:

```
'GSVECM cntl     fixed.cntl.3'
'GSVECM 2        ((1 20 30)(1 40 50))'
'GSVECM 2        .twodim.'           /* twodim is a 2 by 3 array       */
```

Part 2: Installation

Overview of GDDM-REXX installation

This section describes how to install GDDM-REXX; it follows the same style as the *GDDM Installation and System Management for VM* manual, and assumes that you are familiar with the installation procedure for GDDM/VM. You will need to have a copy of that manual to refer to while you are following the instructions in this section.

Things you need to do are indicated by bullets, thus •.

If you are using INSTFPP there are no special considerations for GDDM-REXX.

System and subsystem hardware and software: GDDM-REXX can be used on the same hardware and software as GDDM/VM (see the *GDDM Release Guide* for details), except that you will need to use VM/SP Release 4 or a later release. GDDM/graPHIGS calls are not supported through GDDM-REXX.

The installation of GDDM-REXX consists of the following steps:

Step 1 Preinstallation planning:
   Check that you have installed GDDM/VM.
   Look in the Program Directory for possible updates to this manual.
   Determine space requirements and plan use of storage.
   Check for prerequisites, known errors, and so on.
Step 2 Mount the tape, and read in and run the installation EXEC.
Step 3 Create GDDM-REXX discontiguous saved segment (DCSS) if required.
Step 4 Test the installation.
Step 5 Provide suitable EXECs for users.
Step 6 Inform your users about GDDM-REXX.

Step 1: Preinstallation planning

This section includes estimating storage requirements, and checking that you are ready to install GDDM-REXX.

Storage requirements and capacity planning: You will require sufficient storage on the disk on which you intend to install GDDM-REXX; we recommend that you use the same disk for GDDM-REXX that you used for GDDM/VM; if necessary, expand this disk.

GDDM-REXX occupies approximately 110 4K-byte CMS storage blocks; that is, about one cylinder of 3380 storage, or 200 FBA storage blocks.

GDDM-REXX itself needs storage in which to reside during execution. The code size of GDDM-REXX is approximately 40K bytes.

Your EXECs that use GDDM-REXX will normally reside on your A-disk or other disk storage, and execute in your virtual machine; no special guidance is given for them in this manual. Space requirements will depend on whatever EXECs you plan to develop or use.

- ***Instructions for preinstallation planning:***

1. Check that you have a copy of the *GDDM Installation and System Management for VM* manual handy. You will need it to check your system requirements for GDDM/VM, and for other background information.

2. Check that your VM/SP system is at least at Release 4 level.

3. Check that you have installed GDDM/VM Version 2 Release 1 or later; if you have not yet done this, do it first.

4. Ensure that you have the GDDM-REXX tape available.

5. Check in the GDDM-REXX program directory to see if it contains any corrections to this manual. They are listed under the heading "Updates to *GDDM-REXX Guide.*"

6. Check with the Preventive Service Planning (PSP) "bucket" for late information about installation or necessary fixes. The name of the PSP bucket is given in the program directory. (Ask your IBM service representative if you do not understand this.)

7. Check that you have sufficient disk space (see above) on the disk you will use to install GDDM-REXX. You might find it convenient to use the disk that you used for GDDM/VM.

What IBM supplies: IBM supplies GDDM-REXX and its associated files on tape. The program number is 5664-336. The tape can be supplied in 6250 or 1600 bpi format or formatted for 3480 tape cartridge. Feature codes used are 5870 for 1600 bpi format, 5871 for 6250 bpi format, and 5872 for 3480 tape cartridge. All files on tape are in VMFPLC2 DUMP format. With each tape you are supplied a program directory.

Tape contents: The tape contains the following files:

File no. 1	15664336 011005 program identifier 15664336 installation EXEC
File no. 2	15664336 MEMO for GDDM-REXX, relevant for installation using INSTFPP
File no. 3	Sample and utility files
File no. 4	Null file
File no. 5	GDDM-REXX object code

Step 2: Run the installation EXEC

To install on VM you load an installation EXEC from the distribution tape onto the installation disk, and then use the EXEC to complete the installation.

- ***Instructions for running the installation EXEC:***

1. Log on to a virtual machine, and access the disk you used for GDDM/VM (you need a read-only link to this disk, unless it is to be used also for GDDM-REXX, in which case it must be read/write).

2. Ensure that you do not have any other disks accessed containing previously installed releases of GDDM. If you have, release them.

3. Mount the distribution tape on virtual tape unit 181 (as described in *VM/SP Operator's Guide*).

4. If you are using INSTFPP, refer to the *VM/SP Installation Guide* for what to do next.

5. Enter the following commands at the terminal (system responses are shown in italics):

```
VMFPLC2 REW
  R;
VMFPLC2 LOAD * * filemode
        (where filemode is the mode of the disk you are installing on)
  LOADING.....
  15664336  011005  filemode
  15664336  EXEC  filemode
  END-OF-FILE OR END-OF-TAPE
  R;
```

6. Execute the installation EXEC with the command: `15664336`. You will be prompted for the filemode of the installation disk, and to enter a letter to select the default language for error messages. (The distribution tape contains the files for all supported national languages – there is no separate NL tape.)

Step 3: Create GDDM-REXX discontiguous saved segment

To reduce system overheads you may want to create a saved segment (DCSS) that contains most of the GDDM-REXX code. You should read the section entitled "Create GDDM discontiguous saved segments (DCSS)" in the *GDDM Installation and System Management for VM* manual to help you decide whether you should create a DCSS for GDDM-REXX. You may also need to read the *VM/SP Planning Guide and Reference* and the *VM/SP System Programmer's Guide* for further information. Most of the considerations are the same, and only the differences are noted below:

- The EXEC that enables you to create a DCSS for GDDM-REXX is called ERXBLSEG.

- The name of the saved segment is ERXRX110.

- You will need to calculate the starting address of the saved segment, depending on other saved segments that may be used at the same time as GDDM-REXX. In particular, do not allow GDDM-REXX to overlap the DCSSs for GDDM/VM, and other GDDM programs, such as GDDM-PGF or GDDM-GKS.

If you decide not to create a DCSS for GDDM-REXX, most of the GDDM-REXX code will be loaded as a nucleus extension when a user initializes GDDM-REXX (by issuing the command **GDDMREXX INIT**). This will use approximately 40K bytes of virtual storage in the user's machine.

The GDDMREXX module can also be loaded manually into the nucleus, especially if it is likely to interfere with other modules loaded at address X'20000'. If that is likely to be a problem for your users, you should consider advising them to **NUCXLOAD GDDMREXX**. They will need to do this if EXECs are to be run from CMS subset.

- *Instructions for creating DCSS:*

If you have decided to create a saved segment, continue as follows:

1. Determine at what address the DCSS will start. The address of a DCSS must be higher than the size of any virtual machine that uses it. However, it should not be unnecessarily high, because that may use extra space in VM tables.

   You will require 40K bytes (X'A000').

   Ensure that the DCSS you create will not overwrite any other saved segment for any program related to GDDM that you are likely to use with GDDM-REXX.

2. Generate an entry in the DMKSNT system name table, based on that shown in Figure 1. Use the starting address and size from the preceding step.

   Do not use the information without checking with the system programmer doing the saved segment generation.

```
      System name table entries for use as model for GDDM-REXX
       Do NOT use these figures without checking for overlays

  ERXRX110 NAMESYS SYSNAME=ERXRX110, Name of saved segment
        SYSPGCT=10,                  Count of pages in segment
        SYSPGNM=(2637-2646),         Page numbers used
        SYSHRSG=(166),               Number of segment made up
                                     by these pages
        SYSVOL=gddmvm,               Use whatever name
        SYSSTRT=(015,048),           and values are appropriate
        SYSSIZE=1024K, (ignored; you can always use this value)
        VSYSRES=IGNORE,
        VSYSADR=IGNORE
  This will define a saved segment of 40K (X'A000') bytes
  starting at X'A4D000' (page 2637).
```

Figure 1. Sample system name table entries for GDDM-REXX

3. Log on to a virtual machine with a privilege class that allows you to use the SAVESYS command, and with a virtual machine size at least X'20000' bytes larger than the end of the DCSS. If you use the values in the GDDM-REXX example shown in Figure 1, you will need a storage size of 10832K (X'A94000').

4. IPL CMS and access the disks on which GDDM/VM and GDDM-REXX have been installed.

5. Invoke the EXEC: `ERXBLSEG`.

6. Do **NOT** delete the original copies of any of the GDDM-REXX materials. They may be needed for service as described in *GDDM Installation and System Management for VM*.

Step 4: Test the installation

When you have installed GDDM-REXX, and built your saved segment (if required), you should test that the system works satisfactorily. The simplest way of doing this is by running one of the sample EXECs. If GDDM/VM has not been installed into a saved segment, you will need to issue a `GLOBAL TXTLIB ADMRLIB ADMGLIB` command for the GDDM libraries. (Other GDDM programs may need other text libraries.)

● Enter the command `ERXMODEL` from a terminal that can display graphics. The result is shown on page 15. Your terminal is not suitable for graphics if you see a message saying `ADM0275 W GRAPHICS CANNOT BE SHOWN`.

Step 5: Provide suitable EXECs for users

The EXECs supplied on the installation tape should be sufficient for your users to see how to develop their own programs.

In addition, you might consider providing them with a simple EXEC that links and accesses the disks on which you have installed GDDM-REXX and GDDM/VM, and issues a `GLOBAL TXTLIB` command if you have not installed a saved segment for GDDM-REXX.

Step 6: Inform users about GDDM-REXX

When you have installed GDDM-REXX, you will want to notify your users. Here is a model memorandum that you might consider adapting to your own needs.

MEMO to all system users

New programming tool available for GDDM programmers.

GDDM-REXX is now available on the system. It provides an interpretive interface for accessing GDDM functions through REXX EXECs.

You will need a virtual machine of at least ..... bytes to run GDDM-REXX programs.

GDDM-REXX loads a module at address X'20000'; if this causes any problems with programs you use, try issuing the command **NUCXLOAD GDDMREXX**.

The following IBM manual is available:

> *GDDM-REXX Guide*, order number SC33-0478.

If you encounter any problems running GDDM-REXX EXECs, contact ....

Postinstallation tasks

You should consult the *GDDM Installation and System Management for VM* manual for information. Note that the ADMSERV EXEC can be used to apply corrective service to GDDM-REXX.

You may need to rename some of the sample files if you changed any of the GDDM default filetypes when you installed GDDM/VM.

Installation module directory

These are the files that result from the installation of GDDM-REXX:

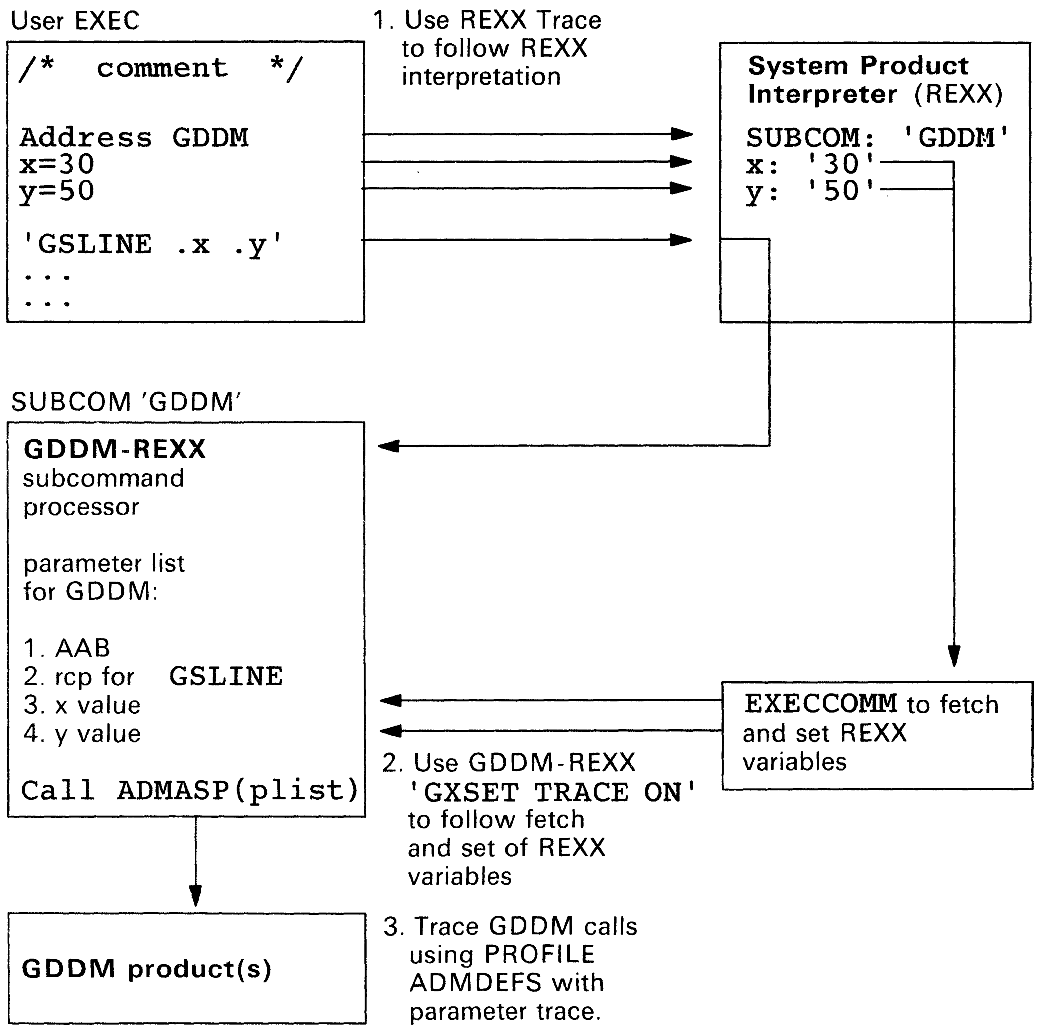
Text library	ERXLIB TXTLIB	
Module files (generated by the installation process; ERXASTUB is generated only if a DCSS is built)	ERXASCOM MODULE ERXASTUB MODULE ERXTMSGA MODULE ERXTMSGB MODULE ERXTMSGH MODULE	ERXTMSGI MODULE ERXTMSGK MODULE ERXTMSGS MODULE GDDMREXX MODULE
Module file (for the default language)	ERXTEMSG MODULE	
Installation material	ERXBLSEG EXEC I5664336 EXEC	I5664336 MEMO I5664336 011005
Sample material	ERXMODEL ADMGDF ERXMODEL EXEC ERXORDD6 ADMGGMAP ERXORDER ADMIFMT ERXORDER EXEC	ERXCHART EXEC ERXMENU EXEC ERXOPWIN EXEC ERXPROTO EXEC ERXTRY EXEC
Utility program	ERXMSVAR EXEC	
Service file (for use with ADMSERV EXEC)	5664336 DATA	

Part 3: Diagnosis

part 3: diagnosis

Here is some more information you may need if you have a problem that is particularly troublesome.

GDDM-REXX is a program that runs in the subcommand environment of REXX. Any command that is not recognized by REXX is passed to the active subcommand environment. To make GDDM-REXX the active subcommand environment, the **Address gddm** instruction is used. Then any symbolic parameters passed are resolved by GDDM-REXX. GDDM-REXX then passes calls to GDDM.



User EXEC
/* comment */
Address GDDM
x=30
y=50
'GSLINE .x .y'
...
...
1. Use REXX Trace to follow REXX interpretation
System Product Interpreter (REXX)
SUBCOM: 'GDDM'
x: '30'
y: '50'
SUBCOM 'GDDM'
GDDM-REXX subcommand processor
parameter list for GDDM:
1. AAB
2. rcp for GSLINE
3. x value
4. y value
Call ADMASP(plist)
EXECCOMM to fetch and set REXX variables
2. Use GDDM-REXX 'GXSET TRACE ON' to follow fetch and set of REXX variables
GDDM product(s)
3. Trace GDDM calls using PROFILE ADMDEFS with parameter trace.

Figure 2. Data flow and trace facilities. Control and data flow are illustrated along with methods of tracing the values sent between REXX and GDDM.

You may need to consult the *VM/System Product Interpreter Reference* manual or the *VM/SP System Programmer's Guide* for further information about subcommand concepts and the REXX **Address** instruction. You should find help with GDDM problems in the *GDDM Diagnosis and Problem Determination Guide.*

When you are trying to find the source of an error, there are three different types of tracing that you can use: REXX, GDDM-REXX, and GDDM. Figure 2 shows how they relate to one another.

Here is a simple REXX EXEC and examples of trace output for it.

```
/* Example of tracing by REXX, GDDM, and GDDM-REXX                    */

Trace r                            /* Start REXX tracing              */
Address command 'GDDMREXX INIT'
Address gddm
s='abcde'
x=70
y=60
'GXSET TRACE ON TIME'              /* Start GDDM-REXX tracing         */
'GSCHAR 50 50 5 .s'
'GSMOVE 40 60'
'GSLINE .x .y'
'GXSET TRACE OFF'                  /* Stop GDDM-REXX tracing          */
'ASREAD . . .'
Address command 'GDDMREXX TERM'
Trace off                          /* Stop REXX tracing               */
Exit
```

The REXX and GDDM-REXX traces were spooled to the console using the CP command `SPOOL CONSOLE START`. Here is part of the output:

```
      4 *-* Address command 'GDDMREXX INIT'
        >>>   "GDDMREXX INIT"
      5 *-* Address gddm
      6 *-* s='abcde'
        >>>   "abcde"
      7 *-* x=70
        >>>   "70"
      8 *-* y=60
        >>>   "60"
      9 *-* 'GXSET TRACE ON TIME'             /* Start GDDM-REXX tracing
 */
        >>>   "GXSET TRACE ON TIME"
 ERX0000 I TIME STAMP: 09/11/86 14:52:04.743196
 ERX0000 I "GXSET TRACE ON TIME"
     10 *-* 'GSCHAR 50 50 5 .s'
        >>>   "GSCHAR 50 50 5 .s"
 ERX0000 I Var fetch: s = "abcde"
 ERX0000 I TIME STAMP: 09/11/86 14:52:06.474704
 ERX0000 I "GSCHAR 50 50 5 .s"
     11 *-* 'GSMOVE 40 60'
        >>>   "GSMOVE 40 60"
 ERX0000 I TIME STAMP: 09/11/86 14:52:06.521208
 ERX0000 I "GSMOVE 40 60"
     12 *-* 'GSLINE .x .y'
        >>>   "GSLINE .x .y"
 ERX0000 I Var fetch: x = "70"
 ERX0000 I Var fetch: y = "60"
 ERX0000 I TIME STAMP: 09/11/86 14:52:06.562355
 ERX0000 I "GSLINE .x .y"
     13 *-* 'GXSET TRACE OFF'                 /* Stop GDDM-REXX tracing
 */
        >>>   "GXSET TRACE OFF"
```

There is more about REXX tracing in the *VM/System Product Interpreter Reference* manual.

GDDM tracing was specified by a PROFILE ADMDEFS file containing this entry (note the space at the start):

```
     DEFAULT TRCESTR='IF API THEN PARMSF TIME'
```

You can find more information on GDDM tracing in the *GDDM Diagnosis and Problem Determination Guide*. The GDDM trace went to a file called ADM00001 ADMTRACE A1, part of which is shown below.

```
/* header and defaults table sections omitted for clarity          */

 00000001 01 CPNIN  SPINIT ('00050000'X) - SPI SPECIAL INIT
 PTRACE    2 CHAR   '
 PTRACE    2 CHAR   '   '
 TIME STAMP  11 SEP 1986 14:52:04  (53524 Seconds)
 00000034 01 CPNOUT SPINIT ('00050000'X) - SPI SPECIAL INIT
 PTRACE    2 CHAR   ---INPUT ONLY PARAMETER-----
 TIME STAMP  11 SEP 1986 14:52:04  (53524 Seconds)

 00000035 01 CPNIN  GSCHAR ('0C0C0500'X) - CHARACTER STRING AT
 PTRACE    2 FLOAT              50
 PTRACE    3 FLOAT              50
 PTRACE    4 DIM                 5
 PTRACE    5 CHAR   'abcde'
 TIME STAMP  11 SEP 1986 14:52:04  (53524 Seconds)
 00000409 01 CPNOUT GSCHAR ('0C0C0500'X) - CHARACTER STRING AT
 PTRACE    2 FLOAT  ---INPUT ONLY PARAMETER-----
 PTRACE    3 FLOAT  ---INPUT ONLY PARAMETER-----
 PTRACE    4 DIM    ---INPUT ONLY PARAMETER-----
 PTRACE    5 CHAR   ---INPUT ONLY PARAMETER-----
 TIME STAMP  11 SEP 1986 14:52:06  (53526 Seconds)

 00000410 01 CPNIN  GSMOVE ('0C0C0400'X) - MOVE TO
 PTRACE    2 FLOAT              40
 PTRACE    3 FLOAT              60
 TIME STAMP  11 SEP 1986 14:52:06  (53526 Seconds)
 00000413 01 CPNOUT GSMOVE ('0C0C0400'X) - MOVE TO
 PTRACE    2 FLOAT  ---INPUT ONLY PARAMETER-----
 PTRACE    3 FLOAT  ---INPUT ONLY PARAMETER-----
 TIME STAMP  11 SEP 1986 14:52:06  (53526 Seconds)

 00000414 01 CPNIN  GSLINE ('0C0C0401'X) - LINE TO
 PTRACE    2 FLOAT              70
 PTRACE    3 FLOAT              60
 TIME STAMP  11 SEP 1986 14:52:06  (53526 Seconds)
 00000421 01 CPNOUT GSLINE ('0C0C0401'X) - LINE TO
 PTRACE    2 FLOAT  ---INPUT ONLY PARAMETER-----
 PTRACE    3 FLOAT  ---INPUT ONLY PARAMETER-----
 TIME STAMP  11 SEP 1986 14:52:06  (53526 Seconds)

/* followed by entries for ASREAD and program termination          */
```

Part 4: Reference

part 4: reference

Summary

Statements and Facilities	
GDDM calls	GDDM Base, GDDM-PGF, and GDDM-GKS are available to give full screen support for alphanumerics, graphics, image, mapping, and charting. GDDM-REXX accesses GDDM calls through the Call Descriptor Table (see the "Call format descriptor module" appendix in the *GDDM Base Programming Reference, Volume 2*). So you can use any of the GDDM calls for which a request control parameter (RCP) is defined, except as noted in "Restrictions and differences" on page 53.
REXX statements	REXX language is available through the System Product Interpreter, and gives high-level language functions and access to CP and CMS environments.
Command	
GDDMREXX	Loads or terminates GDDM-REXX, or displays version number in use.
Subcommands	
GXGET AAB	Obtains the address of the currently used Application Anchor Block (AAB). (Used in conjunction with **FSINIT** and **GXSET AAB**.)
GXGET CDT	Extracts a string of bytes which contains the GDDM call descriptor table (CDT) entry for a given GDDM call. (This is in an encoded form.)
GXGET LASTMSG	Extracts the text of the last error message.
GXGET MSG	Extracts the current state and level of message display.
GXGET NAMES	Extracts a string containing all the GDDM call names (beware, there are several hundred).
GXGET TRACE	Extracts the current state of trace control.
GXSET AAB	Establish the given AAB as current.
GXSET MSADS	In mapping, moves data to the GDDM application data structure (ADS) from the variables that make up the REXX application data structure produced by the ERXMSVAR EXEC.
GXSET MSG	Enables/disables display of messages at the specified severity level or higher.
GXSET MSVARS	In mapping moves data to the REXX application data structure from the GDDM application data structure.
GXSET TRACE	Enables/disables statement and variable fetch/set trace.
Utility EXEC	
ERXMSVAR	Creates a REXX application data structure for use with mapped alphanumerics.

Restrictions and differences

Restrictions	**Notes**
No CHART call	Use the CS... calls that give an improved programming interface to the Interactive Chart Utility (ICU). (See also the sample ERXCHART EXEC.)
No SPINIT call	GDDM-REXX does not support programs that explicitly use the GDDM system programmer interface.

Differences	**Notes**
Array parameters	Array parameters are treated more strictly in GDDM-REXX than they are in other high-level languages. In particular, arrays must be multi-dimensional when they describe lists of lists of values (for example, in the call `'CHBAR 3 2 .ARRAY.'`, the array must be two-dimensional with three sets of two values). Take care with calls for which GDDM-REXX requires an array of strings, where other languages may accept all values concatenated in one string. Such calls include `CHXLAB` and `CHYLAB`. See "Difficult calls" on page 36 for other cases.
FSINIT call	Special action taken. Not needed in normal use; creates a new instance of GDDM. See "Multiple instances of GDDM and GDDM-REXX" on page 31.
FSTERM call	Special action taken. Not needed in normal use; terminates the current instance of GDDM. See "Multiple instances of GDDM and GDDM-REXX" on page 31.
Mapping	The ERXMSVAR EXEC, and two subcommands `GXSET MSVARS` and `GXSET MSADS` are provided so that maps created with GDDM-IMD can be used in REXX EXECs through GDDM mapping calls.
Reentrant support	Reentrant support is provided by means of subcommands that extract and set the anchor block, rather than by a separate set of reentrant calls. The commands are `GXGET AAB` and `GXSET AAB`.
PA2 as escape to subset	By default in GDDM PA2 acts as an escape into subset mode. If CMS subset is entered in this way from a GDDM-REXX EXEC, it is not possible to use further EXECs that use GDDM-REXX. See "Producing reliable, safe, independent EXECs" on page 33 to discover how to give controlled access to CMS subset.
GDDM-REXX EXECs in subset mode or invoked from other programs	If you plan to use GDDM-REXX EXECs from CMS subset mode, or if your GDDM-REXX EXECs are likely to be invoked from other programs, the GDDM-REXX command module should be loaded in the nucleus. See "Using GDDM-REXX from CMS subset or other programs" on page 34 for further details.

GDDM call syntax

In GDDM-REXX, a GDDM call takes the form of the call name followed by a list of parameters. Parameters are separated by blanks both from the call name and from other parameters. Parts of the call can be enclosed in either single or double quotes to prevent processing by REXX.

Methods of passing parameters

Parameters may be REXX variable names or literal values. Using a REXX variable name is known as *calling by name*. Using a literal value is known as *calling by value*. The methods can be mixed within a single GDDM call. When parameters are passed by name, they must be preceded by a dot.

```
'GSLINE .x .y'                  /* by name                     */
'GSLINE 10 20'                  /* by value                    */
'GSLINE .x 20'                  /* mixture of name and value   */
'ASCPUT .id .length .words'     /* by name with character string  */
'ASCPUT  1    5   "Hello"'      /* by value with character string */
```

The names passed can be any type of REXX variable; the correct conversion will be made to the type required by GDDM.

Note that you cannot pass a variable that contains a string of parameters. For example, you cannot use
`parmstring='10 20'; 'GSLINE .parmstring'.`

Values that can be passed

The values passed can either be integer, floating point, or string. Strings may be of variable length or of fixed length. Fixed-length strings are sometimes called tokens. Values allowed are shown in the table.

Type	Values allowed
integer	Range from −2**31 to 2**31−1. Floating point and a decimal point are allowed provided any fractional digits are zero. Take care with negative values. Include quotes `'GSCOL -1'` or parentheses if outside quotes `'GSCOL' (-1)`. Note that the REXX `NUMERIC DIGITS` instruction has no effect on the range. Examples: `1` `21` `1234` `-1`
float	Floating point or decimal notation. Maximum and minimum restricted to System/370 short floating point form. However, GDDM graphics calls limit values to a smaller range (absolute values of nonzero parameters in the range 1.0E−18 to 1.0E18). Examples: `1.3` `1e+3` `1E-3` `1.0e3`
strings (fixed length)	Most are names of GDDM objects and should be coded in uppercase. GDDM does not recognize the lowercase versions. Example: `ADMUUKSF`
strings (variable length)	Enclosed in double quotes or single quotes. If GDDM calls are in single quotes (as advised), double quotes should be used. If you want a quote displayed or printed, use it twice if it is already used as a string delimiter. Examples: `"String"` `'string'` `'don''t'` `"He said, ""Don't"""`

Types of parameters

Parameters must be either scalars (a single value) or arrays (a list of values with defined dimensions).

Passing scalar parameters

When passing by name, assign the value to the variable and use the name preceded by a dot in the call. When passing by value, include the value in the call; for character strings enclose the string in double quotes.

```
x=50.1
y=10
l=16
words="This is a string"
'GSCHAR .x .y .l .words'              /* call by name          */
'GSCHAR 50.1 10 16 "This is a string"' /* call by value         */
```

String or token parameters may be entered without quotes if, and only if, the string contains none of the special characters: blank, single or double quote, left or right parenthesis, or the shift-out SO character. In addition, any token or string that begins with a dot must be enclosed in surrounding quotes.

GDDM-REXX lets you pass a character string that contains DBCS (double-byte character strings). They must be between Shift-out and Shift-in (SO/SI) brackets. (SO = X'0E' and SI = X'0F'.) It is an error if an unmatched SO occurs in a string.

Strings containing these special characters must be enclosed in matching quotes. This is because they can cause parsing to be interrupted. For example:

```
s = 'a b c'                     /* string containing blanks       */
'GSCHAR 30 50 5' s              /* "s" is evaluated - the command */
                                /* passed to GDDM after evaluation*/
                                /* is 'GSCHAR 30 50 5 a b c'      */
                                /* which has too many parameters  */
```

However, you can safely omit the quotes in some circumstances, like:

```
'CHXLAB 12 3 (JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC)'
```

Character strings that contain blanks or DBCS characters can be passed "by name" without the need for any special GDDM-REXX string delimiters. However, the system interpreter requires an `OPTIONS ETMODE` statement before the SO/SI characters in literal strings or comment statements.

Passing array parameters

Array parameters can be passed in the following ways:

1. By name using a REXX stemmed variable. The array will be taken from the member .1 or .1.1 (and so on).

   ```
   xarray.1=10; xarray.2=20        /* and so on                     */
   yarray.1=10                     /* and so on                     */
   'GSPLNE 3 .xarray.  .yarray.'   /* by name with stemmed variables */
                                   /* Note dot after stemmed name    */
   ```

2. By name using a prefix. REXX variables of the form `prefix1`, `prefix2`, for one dimension and `prefix1.1`, `prefix1.2` for two dimensions will be looked for and their values used. The variables with these new names are formed into a list and passed as an array to GDDM. For example:

   ```
   varx1=10; varx2=8; varx3=5      /* and so on                     */
   'GSPLNE 3 .varx .vary'
   ```

   is passed to GDDM as

   ```
   'GSPLNE 3 (.varx1 .varx2 .varx3) (.vary1 .vary2 .vary3)'
   ```

3. By name or value enumerated in list between parentheses. For example:

   ```
   bot=5; mid=25
   'GSPLNE 3 (10 20 .bot)(20 .mid 30)'/* by name and value. Each member */
                                   /* enumerated in parentheses      */
   ```

   or, for a two-dimensional array:

   ```
   'CHBAR 3 2 ((10 40) (20 50) (30 60))'
   ```

4. By stem or prefix names in parentheses; these represent columns in a two-dimensional array. For example:

```
barray.1=10; carray.1=40
barray.2=20; carray.2=50
barray.3=30; carray.3=60
'CHBAR 3 2 (.barray. .carray.)'     /* dimensioned values placed in   */
                                    /* parentheses when required      */
                                    /* array needs more dimensions    */
```

Note that this works only for columns, not for rows. The listed values to achieve the same results would be either of these forms:

```
'CHBAR 3 2 ((.barray.1 .carray.1) (.barray.2 .carray.2) (.barray.3 .carray.3))'

'CHBAR 3 2 ((10 40) (20 50) (30 60))'
```

Handling of short or long parameters

The method of handling parameters that do not exactly match the specifications varies according to the type of parameters. All mismatches not described below are treated as errors.

Parameters that are too short

Strings	Padded with blanks to the required length. This may be the length specified in the call or, for tokens, the length specified by GDDM.
Too few array elements	Extra members are generated. When the array is passed by name, the REXX variables are searched for, and it is an error if they do not exist. When the array is passed by value, extra values are added as necessary. They are zero for floating point and integer parameters, and blanks for strings and tokens.
Too few array dimensions	The item is rescanned to produce the correct number of dimensions. If the array is passed by name, the necessary suffixed names are generated, and it is an error if they do not exist.

If the array is passed by value, additional values are generated using the following rules:

1. If the size of the missing dimension is explicitly given, the list will be scanned this number of times. If an individual list entry is a value, that value will be reused. For example:

   ```
   'ASDFMT 2 . (1 2 3 4 5 6)'  /* vector used twice              */
   ```

   becomes

   ```
   'ASDFMT 2 6 ((1 2 3 4 5 6) (1 2 3 4 5 6))'
   ```

   If it is a name, it is suffixed:

   ```
   'ASDFMT 1 6 (.f .r 3 4 5 6)'/* extra dimension with suffixes */
   ```

   is processed as

   ```
   'ASDFMT 1 6 ((.f1 .r1 3 4 5 6))'
   ```

2. If the size of the missing dimension is not explicitly given, that dimension is defaulted to 1, and the result is a 1-by-n array. For example:

   ```
   'ASDFMT . . (1 2 3 4 5 6)'  /* 1-by-6 array generated        */
   ```

   becomes

   ```
   'ASDFMT 1 6 ((1 2 3 4 5 6))'
   ```

3. If the list is of two dimensions less than required, the list is scanned the required number of times to produce, for each of the required number of rows, the values in the columns. If the entry is a literal value it is reused. For example:

   ```
   'ASDFMT 2 6 1'              /* rescanning gives 2-by-6 array */
   ```

   becomes

   ```
   'ASDFMT 2 6 ((1 1 1 1 1 1) (1 1 1 1 1 1))'
   ```

   Omission of either size parameter defaults to the value 1.

Parameters that are too long

Strings	Truncated to the required length, with an error message.
Too many array elements	Extra members are ignored.
Too many array dimensions	This is treated as an error and a message is given.

Omitting parameters

Parameters can be replaced by dots if they are:

1. Returned values that the program is not interested in
2. Lengths that GDDM-REXX can discover from your input
3. Array dimensions that GDDM-REXX can discover from your input.

```
'ASREAD . . .'                  /* omit returned values          */
'ASCPUT 1 . "Hello"'            /* omit length you are passing   */
'CHBAR  . . ((1 2) (3 4) (5 6))'  /* omit array defining counts  */
```

If two or more parameters depend on an omitted length or count value, the first one from which the value can be determined is used, and subsequent parameters are processed with this "discovered" value.

Finding syntax from reference sources

The parameter syntax of GDDM calls can be found by use of the sample program ERXPROTO or in the GDDM programming reference manuals or summaries.

Interdependent parameters, array dimensions, string lengths

Many GDDM calls have interdependent parameters where earlier lengths and counts describe the lengths of strings and count of elements in arrays. These calls are described in the *GDDM Base Programming Reference*:

```
GSCHAR(x,y,length,string)
```

where `length` is the length of the `string`; and

```
GSPLNE(count,xarray,yarray)
```

where `count` is the number of elements in each of the two arrays.

Where there is one such dependency, the length is the length of the string, or the count is the number of elements in the array.

Where there are two dependencies, strings are given in a one-dimensional array of strings of the given length; numbers are given in a two-dimensional array, with the first count specifying the number of groups and the second specifying the number of elements in each group.

```
CHXLAB(count,length,text)
```

`text` is an array of `count` strings each of the number of characters in `length`.

```
CHBAR(components,count,y-values)
```

`y-values` is a two-dimensional array with `components` rows and `count` columns.

This is shown explicitly for GDDM/VM and GDDM-PGF in the reference summaries and for GDDM-GKS in the reference manual. The same information is available from the ERXPROTO EXEC.

```
'CHXLAB cntl len2 char.cntl.len2'  /* ERXPROTO forms for the two     */
'CHBAR cntl cnt2 float.cntl.cnt2'  /* calls described above          */
```

Parameter syntax in ERXPROTO

ERXPROTO produces output of the following form. (The example is used to show complete syntax; there is no GDDM call of this type.)

```
'callname cntl cnt2 len3 .float intg.cntl.cnt2 char.len3 char.8'
```

`cntx`	These counts are used as array dimensions. The number that follows them is their position in the string.
`lenx`	These lengths are lengths of character strings. The number that follows them is their position in the string.
`.float`	The parameter is floating point. The dot that precedes it means that it is returned by GDDM.
`intg`	The parameter is an integer.
`char`	The parameter is a character string.
`.cntl.cnt2`	These are array dimensions -- see below for rules to deduce dimensions and sizes.
`.len3`	This is the length of the character string it follows. If the parameter is an array, it is the length of each string in the array.
`.8`	This is a constant value required by GDDM. If it is the last item that starts with `char` (as in this case) it is a length, in other cases it is an array dimension.

Here is a set of rules that let you produce valid calls from the ERXPROTO syntax.

1. Look for `cnt` values that are array dimensions, and calculate the values you will need for them.

   ```
   'CHBAR cntl cnt2 float.cntl.cnt2'  /* two sets so two dimensions     */
                                      /* in example cntl=2,cnt2=3       */
   'GSCHAR float float len3 char.len3'/* none .len3 is a length         */
   'ASREAD .intg .intg .intg'         /* none                           */
   ```

   For a by-name array you would enter values `array.1.1=10` to `array.2.3=60` (using the values you needed) and use the parameter `.array.` (with a closing dot).

2. Look for any array parameters and work out the correct dimensions. Array parameters are followed by `.cnt`, for example `float.cntl`. There is one dimension for each following `.cnt`. That means one set of brackets for each following dot if you are listing the array elements in the call.

   ```
   'CHBAR  2  3  ((n n n) (n n n))'   /* cntl and cnt2, two dimensions  */
                                      /* three (cnt2) elements in each  */
                                      /* inner parenthesis, two (cntl)  */
                                      /* sets of inner parentheses. Two */
                                      /* levels of nested parentheses,  */
                                      /* one for each count             */
   'GSCHAR float float len3 char.len3'/* no arrays no action            */
   'ASREAD .intg  .intg  .intg'       /* none                           */
   ```

3. Look for the character strings and fill in the length values.

   ```
   'GSCHAR float float 4  "ABCD"'     /* Characters are ABCD so length  */
                                      /* is four                        */
   ```

4. Fill in the float or integer values or variable names. These are integers where the parameter says **intg**, any form of number where it says **float**, and character strings where it says **char**.

```
'CHBAR 2  3 ((3 4 5) (5 6 7)) '     /* put in array values              */
'GSCHAR 10.5  50    4  "ABCD"'      /* fill in values                   */
'ASREAD .type  .val   .count'       /* fill in all values               */
```

Parameter syntax in the reference manuals

The reference manuals present GDDM calls using a syntax that has parentheses around the parameters and commas between them, for example:

```
GSLOAD (name,count1,opt-array,seg-count,count2,descriptor)
```

The four following steps will produce calls in the correct syntax for GDDM-REXX:

1. Omit the parentheses around the parameters and replace the commas with blanks:

   ```
   GSLOAD name count1 opt-array seg-count count2 descriptor
   ```

2. Use a valid REXX variable name for each parameter, optionally using stemmed variable names for array parameters:

   ```
   GSLOAD name count1 opt_array. seg_count count2 descriptor
   ```

3. Place a period '.' before each parameter name. This indicates that the parameter is to be passed "by name":

   ```
   GSLOAD .name .count1 .opt_array. .seg_count .count2 .descriptor
   ```

4. Surround the entire statement with single or double quotes (this is to ensure that the interpreter passes the complete call to GDDM-REXX without attempting substitution):

   ```
   'GSLOAD .name .count1 .opt_array. .seg_count .count2 .descriptor'
   ```

Now, create REXX assignment statements for all variables which are defined as *specified by user*, that is variables that the program passes to GDDM. Place them before the GDDM call; for example:

```
name='MYGDF'                /* name of the ADMGDF file to be loaded  */
count1=2                    /* number of elements in opt_array.      */
opt_array.1=22              /* starting segment number to be assigned */
opt_array.2=2               /* accommodate to current window size    */
count2=110                  /* return up to 110 bytes of descriptor  */
'GSLOAD .name .count1 .opt_array. .seg_count .count2 .descriptor'
```

After the **GSLOAD** is performed, the REXX variables **seg_count** and **descriptor** will contain the *returned by GDDM* values.

Rules for deducing the syntax for string and array parameters from the *GDDM Base Programming Reference* or *GDDM-PGF Programming Reference* manuals are as follows:

1. Look for any dependencies between parameters — the parameter names *length* and *count* always show dependencies but they are not the only ones. You must read the parameter descriptions to be sure.

2. When you have found a dependency, check whether the item it describes is numerical or a character string.

3. If there is one dependency, it is the number of elements in a one-dimensional array for numbers, or the length for a character string.

4. If there is more than one dependency:
   For numerical parameters: The number of dependencies specifies the number of dimensions; the first item in the list becomes the number of elements in the first dimension, the second item becomes the number of elements in the second dimension, and so on.
   For string parameters: The number of dependencies is one more than the number of dimensions of the array. The last dependency is the length of each of the strings in the array. Prior dependencies specify the number of elements in each dimension.

GDDMREXX command

```
GDDMREXX INIT [( [NODCSS] [LANG x] [)]]
```

Initializes GDDM-REXX.

NODCSS Prevents the discontiguous saved segment (DCSS) being loaded, if there is one. Instead a copy of all of GDDM-REXX is loaded into user storage. If the DCSS is already in use, NODCSS is rejected with an error message. To unload the DCSS you must issue the `GDDMREXX TERM` command and re-initialize. A DCSS is an area of CMS storage available to many users.

LANG x Loads GDDM-REXX message text in the appropriate language (module name ERXTMSGx). It has no effect on messages from GDDM or elsewhere. The meaning of the letters is shown below. Some of them may not be available in your installation. `x` is one of the following letters:

A	U.S. English	H	Hangeul	K	Kanji
B	Brazilian	I	Italian	S	Spanish

Note that if a double-byte character set language is used for GDDM or GDDM-REXX languages, we recommend that you operate with `GXSET MSG OFF`, and use GDDM to display the error messages that you extract with `GXGET LASTMSG`.

```
GDDMREXX TERM [(ALL[)]]
```

Terminates GDDM-REXX and frees the storage used by it. Note that termination of GDDM-REXX (and, hence, GDDM) also implicitly occurs at CMS command ready.

ALL Terminates all instances of GDDM-REXX.

```
GDDMREXX VERSION [(STACK|LIFO|FIFO[)]]
```

Returns the version and release level, product number, date, and copyright notice of the copy of GDDM-REXX being used. The options allow the use of this command within REXX EXEC files. If they are omitted, the response is returned to the terminal.

STACK The information is queued onto the CMS stack behind any items that are already on the stack.

LIFO Last in/first out. The information is pushed onto the CMS stack before any items that are already on the stack.

FIFO First in/first out. Same as STACK.

GXGET subcommand

The GXGET subcommand is used to extract information from GDDM-REXX.

```
GXGET AAB .aabtoken
```

Extracts a token that relates to the current application anchor block (AAB). This can later be restored by a **GXSET AAB** subcommand, which will enable the correct instance of GDDM. (Used in conjunction with **FSINIT** and **GXSET AAB**.)

.aabtoken	Variable in which the token relating to the current AAB is returned. You should not tamper with this token in any way. See "Multiple instances of GDDM and GDDM-REXX" on page 31 for more information.

```
GXGET CDT .name .entry
```

Gives the contents of the GDDM call descriptor table (CDT) in a byte string. The CDT is described in the *GDDM Base Programming Reference*. The bytes need to be interpreted – a method of doing this is shown in the sample ERXPROTO EXEC.

.name	The name of the variable containing the name of the call for which the CDT entry is required. This may be coded as a literal, thus: **'GXGET CDT GSLOAD .gslcdt'**
.entry	A variable in which the CDT byte string will be returned.

```
GXGET LASTMSG .msg
```

Gives text of the last error.

.msg	Variable in which the text of the last error message is put. If there have been no previous error messages, the string is empty.

```
GXGET MSG .state .level
```

Gives current state and level of message handling.

.state	Variable in which the state of message handling is returned; its value is ON or OFF.
.level	Variable in which the level of messages shown will be returned. Values are:

0	(informational) messages and above
4	(warning) messages and above
8	(error) messages and above
12	(severe) messages and above.

```
GXGET NAMES .namelist
```

Gives a string containing all the GDDM call names. Note that there are several hundred calls, all of which are included.

.namelist	The variable into which the list of names is placed.

```
GXGET TRACE .state .time
```

Gives the current state and level of trace.

.state	Variable in which the state of tracing is returned. It may be ON or OFF.
.time	Variable in which the state of trace timing is returned. It may be TIME or NOTIME.

GXSET subcommand

The GXSET subcommand is used to pass information to GDDM-REXX. Although this syntax shows variable names, literals may be used instead (except where indicated), as in `'GXSET TRACE ON TIME'`.

```
GXSET AAB .aabtoken
```

Establishes the given AAB as current. Used in conjunction with `GXGET AAB`.

.aabtoken	Variable containing the anchor block to be established as current, so that the associated instance of GDDM will be used. See "Multiple instances of GDDM and GDDM-REXX" on page 31 for more information.

```
GXSET MSADS .mapgrp .map .prefix .ads
```

Moves data to the user's application data structure, which must have been created with the ERXMSVAR EXEC. Use before output when using mapped alphanumerics.

.mapgroup	Name of the mapgroup that contains the map
.map	Name of the map
.prefix	The prefix specified in the ERXMSVAR EXEC. This is used as a stem to give variable names that are suitable for use with REXX. Note that the dot is required.
.ads	The name of the application data structure variable. See "Mapping" on page 28 for more information.

```
GXSET MSG .state [.level]
```

Enables/disables display of messages at specified severity level or higher. The default is that the state is **ON** with a level of 4, meaning that warning messages and those of a greater severity are shown.

.state	Sets message handling state; it may be ON or OFF.
.level	Sets message handling level; it may be:

0	echo (display before execution) user statements and display all messages
4	(warning) messages and above
8	(error) messages and above
12	(severe) messages and above.

```
GXSET MSVARS .mapgrp .map .prefix .ads
```

Moves data from the user's application data structure, which was created with the ERXMSVAR EXEC. Use after input when using mapped alphanumerics.

.mapgrp	Name of the mapgroup that contains the map.
.map	Name of the map.
.prefix	Prefix specified in the ERXMSVAR EXEC. This is used as a stem to give variable names suitable for use with REXX. Note that the dot is required.
.ads	Name of the application data structure. See "Mapping" on page 28 for more information.

```
GXSET TRACE .state [.time]
```

Enables/disables statement and variable tracing.

.state	ON to start tracing, OFF to end tracing.
.time	Optional parameter used with ON. TIME causes a time-stamp record to be produced with the trace record. NOTIME suppresses time-stamping.

ERXMSVAR EXEC

```
ERXMSVAR mapgroupname mapname prefix
```

ERXMSVAR produces initialization statements for REXX variables that are associated with a mapgroup and map. It produces a CMS file named **mapname GDDMCOPY** to include in a mapping program. See "Mapping" on page 28 for more information.

When GDDM Interactive Map Definition has been used to define and generate the map and mapgroup, ERXMSVAR may be executed.

All maps used with GDDM-REXX should be generated with the option to include field names. In GDDM-IMD panel 3.0 specify:

```
FIELD NAMES INCLUDED IN GENERATED MAPGROUP  ==> YES
```

Names of the form `NO_NAME_1`, `NO_NAME_2`, and so on are assumed if field names are not included. This is discussed further below.

mapgroupname	The name of the generated mapgroup. The mapgroup must have the filetype of ADMGGMAP. This is the filetype generated by GDDM-IMD. If the mapgroupname ends with two dots, GDDM will supply the last two characters, which indicate the device class.
mapname	The name of the map.
prefix	The prefix used for the associated REXX variables.

Field naming rules:

1. Field names (including selector and adjunct names) follow the normal GDDM-IMD rules. The names can be seen in the GDDMCOPY file produced by the ERXMSVARS EXEC.

   Field names and adjunct suffixes are converted into names acceptable to REXX, by changing hyphens (–) to underscores (_). Thus, `MY-FIELD` becomes `MY_FIELD`, and if it has the color selector adjunct `COL-SEL`, that becomes `MY_FIELD_COL_SEL`.

   Adjunct suffixes depend on the language you selected when you generated the mapgroup (see the *GDDM Base Programming Reference* for details).

2. If you do not use

   ```
   FIELD NAMES INCLUDED IN GENERATED MAPGROUP  ==> YES
   ```

   GDDM-REXX generates field names as follows:

for fields	`prefix||"NO_NAME_"||field-number`
for adjuncts	`prefix||"NO_NAME_"||field-number||adjunct-suffix`

   where:

`prefix`	is as previously described.
`NO_NAME_`	is the standard name given to all fields.
`field-number`	is the GDDM-IMD sequence number of the field in the map (array indexes, if any, are lost).
`adjunct-suffix`	is the suffix that appears on all adjunct variables. It takes the PL/I form, using underscores.

   For example:

   ```
   X_NO_NAME_3 = "            "
   X_NO_NAME_4 = "      "
   X_NO_NAME_4_COL_SEL = " "
   ```

Sample output: Assuming there were three arrays of fields called PROD, DESC, and COST (as there are in the sample map ERXORDER), the command

```
ERXMSVAR group1 map1 X_
```

generates:

```
/* GDDM-REXX: output from ERXMSVAR EXEC:   . . . 16:41:32           */
/* Initialize structure for MAPGROUP: ERXORDD6 , MAPNAME: ERXORDER */
X_=""
X_PROD.1 = "          "
X_DESC.1 = "                        "
X_COST.1 = "          "
X_QTY.1 = "     "
X_QTY_COL_SEL.1 = " "
X_QTY_COL.1 = " "
X_TOT.1 = "        "
X_PROD.2 = "          "
 ...
 ...                                "
 ...
X_TOTAL = "        "
X_MSG = "                                "
X_ASLENGTH=432 /* length of ADS string */
```

Examples of REXX variable names used for a map without names:

```
X_NO_NAME_3 = "          "
X_NO_NAME_4 = "     "
X_NO_NAME_4_COL_SEL = " "
X_NO_NAME_4_COL = " "
```

Possible pitfalls: If you use a stemmed variable such as **stem.** do not use the same variable names in the EXEC that are used as any field names in your maps, as this could result in unwanted substitutions. For example:

```
title='Overdue orders'
stem.title=title
  ...
```

results in:

```
stem.Overdue orders=Overdue orders
```

The variable in your map **stem.title** will not have been altered.

Error message explanations

When GDDM-REXX detects an error which has severity at or above a specified level, it issues an error message, preceded by an information message which displays the user statement.

The severity level is set by the **GXSET MSG** subcommand or defaults to 4 (warning messages and those of a higher severity). If you request GDDM-REXX tracing by issuing the subcommand **GXSET TRACE ON**, you do not need to use the **GXSET MSG** subcommand to see the trace output.

Messages take the form:

```
ERX0000 I user-statement-text
ERXnnnn s message-text
```

where:

```
ERX  GDDM-REXX prefix
nnnn message number
s    severity character
```

Other messages: You may also get GDDM, REXX, CMS, or CP messages during the running of your EXECs.

GDDM messages (explained in *GDDM Messages*) take the form:

```
ADMnnnn s message-text
```

System Product Interpreter (REXX) messages (explained in *VM/System Product Interpreter Reference*) take the form:

```
DMSREXnnn s message-text
```

where:

```
DMSREX System Product Interpreter prefix
nnn    message number
s      severity character
```

CMS messages (see *VM/SP System Messages and Codes*) take the form:

```
DMSxxxnnn s message-text
```

where:

```
DMSxxx CMS prefix
nnn    message number
s      severity character
```

Disappearing messages: If messages disappear before you have had time to read them and are replaced with a GDDM screen or an empty screen, put an `FSFRCE` call immediately after the `Address gddm` command. Messages are cleared when GDDM opens a device, typically at the first `ASREAD` call. The `FSFRCE` opens the device before any calls have been made.

Return codes: GDDM-REXX commands, subcommands, and GDDM calls will produce return codes to the user's REXX program. With the exception of ERX0001, these return codes are related directly to the severity level indicated in the text of the message as follows:

REXX return code ("rc" variable)	GDDM-REXX severity character in message, and its meaning	
-3		(error ERX0001)
0		(no error found)
0	I	information message
4	W	warning message
8	E	error message
12	S	severe error

ERX0000 I '....user statement....'

ERX0000 I TIME STAMP: '....timing information....'

ERX0000 I Var fetch: '....varname....' = '....value....'

ERX0000 I Var set: '....varname....' = '....value....'

Explanation: This message is issued when 'GXSET TRACE ON' has been executed. The user statement form is produced when a GDDM error occurs, and is normally accompanied by a message starting ADM. The time stamp form is generated when 'GXSET TRACE ON TIME' is specified. "Var fetch" means that a variable has been fetched from REXX and has the value shown. "Var set" means that a variable has been given the value shown and passed to REXX.

ERX0001 E Unknown GDDM function call

Explanation: Follows a REXX message that displays the call. The call may be a mistyping of a GDDM call, or it may be a command that should have been passed to another environment, or it may be an error caused by REXX substitution. For example, the call 'GSCLP' 1 = 1 will send to GDDM-REXX the character string '0'. (This is because concatenation has higher priority than comparison in REXX, and the comparison between 'GSCLP 1' and '1' yields '0'.)
It will also occur if you attempt to use the CHART or SPINIT call.
This message will also occur if you are not using a discontiguous saved segment (DCSS) for GDDM, and have not issued the appropriate GLOBAL TXTLIB command.

ERX0002 E Too few parameters

Explanation: The number of parameters in the call is too small. In particular check that array parameters are correctly coded. See the summary of calls and description of parameters in the *GDDM Base Programming Reference Summary* and the *GDDM-PGF Programming Reference Summary*.

ERX0003 E Too many parameters: '....'

Explanation: The number of parameters in the call is too large. In particular check that array parameters are correctly coded. This error is often caused by a bad parenthesis count.

ERX0004 E Invalid parameter type: '....'

Explanation: A string parameter was unquoted and began with a "(" or ")"; correct it.

ERX0005 E Invalid integer value: '....'

Explanation: The item passed was not an integer. It could have been an uninitialized REXX variable, a character string, or a number that was not an integer.

ERX0006 E Invalid real value: '....'

Explanation: The item passed was not a number. It could have been an uninitialized REXX variable or a character string.

ERX0007 E Invalid parameter: '....'

Explanation: In a GXSET subcommand, the cause could have been: (1) token parameter longer than eight characters, (2) value coded for return parameter (missing the "."), (3) invalid keyword given (too long, misspelled), (4) invalid subverb (MSG, LASTMSG, etc.), (5) incorrect integer syntax, (6) invalid AAB token (for GXSET AAB).
In a GXGET subcommand, the cause could have been: (1) token parameter longer than eight characters, (2) value coded for return parameter (missing the "."), (3) invalid keyword given (too long, misspelled), (4) invalid subverb (MSG, LASTMSG, etc.).

ERX0008 E Parameter rank too large: '....'

Explanation: Array parameter rank (number of dimensions) greater than that expected. For example, a two-dimensional array passed when a one-dimensional array was needed. Either the array was coded with correct syntax, but the rank was too large, or it was coded incorrectly, and GDDM-REXX interpreted it as having the wrong rank.

ERX0009 E Invalid parameter syntax: '....'

Explanation: Check array parameters for the following: (1) token encountered at higher than innermost rank, (2) too many right parentheses, (3) array pre-scan failed: array began with right parenthesis, or found "(" when processing values in a row, or found a value after a ")".

ERX0010 E Missing parameter(s)

Explanation: If the parameter is one that is set by or passed to GDDM the cause could have been: (1) fewer than minimum required number of parameters, (2) subverb missing.

ERX0011 W REXX variable had no value: '....'

Explanation: If the variable is in a parameter that will be passed to GDDM-REXX: (1) For a string parameter, processing continues with the name being used as the value. (2) For a numeric parameter, processing continues until the number conversion fails, producing ERX0005 or ERX0006. If the variable is in a parameter that will be returned to REXX, the statement is not executed.

ERX0012 W REXX variable '....' truncated

Explanation: If the variable is in a parameter that must be passed to GDDM, the truncated string is passed for a character string, and the process is stopped for a numerical parameter. If the variable is in a parameter that has been returned by GDDM, processing continues. One of the following has happened: (1) a numeric parameter was longer than 18 characters, (2) a token parameter was longer than 8 characters, (3) a keyword parameter (e.g. "ON") was longer than 18 characters, (4) a string parameter was longer than required by GDDM.

ERX0013 E Variable name required for return parameter: '....'

Explanation: Check for (1) missing returned-by-GDDM parameter, (2) value specified for returned-by-GDDM parameter (leading "." missing).

ERX0014 E Length must be specified for return parameter: '....'

Explanation: Lengths must be specified for returned parameters, for example in 'ASCGET id length string'. This usually occurs when a user has coded a single dot for the length (indicating omitted length). (A dot coded for the length of a returned numeric array parameter will be treated as a length of 1.)

ERX0015 S Insufficient free storage

Explanation: This message is issued when any request for free storage fails. The user may have defined the virtual machine to be too small to accommodate the data in the application. A CMS message will precede the ERX0015 message:

```
DMSFRE159T INSUFFICIENT STORAGE AVAILABLE TO
SATISFY DMSFREE REQUEST FROM 'addr'
```

The request is conditional and processing continues, however the particular user request which created the error condition will not be executed.

ERX0016 E Return code '....' from EXECCOMM: '....'

ERX0016 E Return code '....' from EXECCOMM: '*******'**

Explanation: The return code will normally be 8, indicating a bad name for a REXX variable. The second insert is the name in question. If the second insert is all *, then the return code is from register 15 upon exit from the EXECCOMM call (EXECCOMM is the means by which nonEXEC programs can set or fetch EXEC variables).
Details of other return codes can be found in the *VM/System Product Interpreter Reference* manual.

ERX0017 E Unknown GDDM-REXX subcommand

Explanation: Invalid subverb discovered.

ERX0018 E Right parenthesis missing
Explanation: An unmatched parentheses in an array parameter has been found (either a numeric array or a list of strings).

ERX0019 E Matching SI character not found in DBCS string
Explanation: SO character found in quoted string parameter, but end-of-statement encountered before finding matching SI character.

ERX0020 E Ending string delimiter missing
Explanation: In a string starting with either ' or ", the end-of-statement was encountered before finding the matching string delimiter.

ERX0098 E Error in descriptor '....' reason '....'
Explanation: Part of GDDM's calling mechanism has been corrupted or unexpectedly changed. This error will occur if the GDDM CDT changes format, or is somehow destroyed or altered in storage.
The first substitution is the parameter number of the entry in the CDT. The second is the type of the error:
1 accumulator rank greater than 3
2 invalid parameter type
3 no matching variant found
4 number of accumulators greater than ten
5 number of parameters greater than 32.

ERX0099 E Unknown message number '....'
Explanation: The GDDM-REXX error handler has been called with an invalid message number. This is a system or GDDM error.

ERX0100 E Invalid parameter: '....'
Explanation: The parameter shown in the message is not recognized by the GDDMREXX command.

ERX0101 E Invalid option: '....'
Explanation: The option shown in the message is not recognized by the GDDMREXX command.

ERX0102 E No function specified
Explanation: No function was specified on the GDDMREXX command.

ERX0103 W GDDM-REXX has not been initialized
Explanation: 'GDDMREXX TERM' or 'GDDMREXX TERM (ALL)' has been requested, but GDDM-REXX is not operational.

ERX0104 E Return code '....' from '....'
Explanation: GDDM-REXX calls some other CMS commands and functions. The command shown in the message gave the reported unexpected return code. Refer to the CMS HELP information for the command in error for more details about the meaning of the return code. Variants are:
NUCXLOAD – the error occurred in the NUCXLOAD command, loading the ERXASCOM nucleus extension.
NUCEXT – the error occurred in the NUCEXT function, querying the ERXASCOM nucleus extension that has just been loaded.
NUCXDROP – the error occurred in the NUCXDROP command, deleting the ERXASCOM nucleus extension.
SUBCOM – the error occurred while defining the GDDM SUBCOM environment.
ATTN – the error occurred while stacking the result of the GDDMREXX VERSION command.

ERX0105 E 'NODCSS' option is invalid. DCSS is already in use
Explanation: The NODCSS option has been specified on the GDDMREXX command, but GDDM-REXX is already initialized using the DCSS. The request for the NODCSS option is ignored.

ERX0106 S GDDM-REXX DCSS is no longer loaded
Explanation: A subcommand has been issued to the GDDM environment, but the GDDM-REXX DCSS which should process the subcommand has somehow been unloaded from the virtual machine.

ERX0107 E ERXASCOM must not be called as a command
Explanation: A user has issued ERXASCOM as a command from the terminal. This is not permitted.

ERX0108 E GDDM-REXX could not locate SUBCOM 'GDDM'
Explanation: 'GDDMREXX TERM' has been issued, and ERXASCOM nucleus extension exists. However, there is no matching GDDM SUBCOM and so there is nothing to terminate.

ERX0109 S GDDM-REXX is not supported on this level of CMS
Explanation: While processing a 'GDDMREXX INIT' command GDDM-REXX found that the level of CMS on the system is not VM/CMS SP-4 or later.

ERX0110 S Recursive entry to GDDM-REXX. Request rejected
Explanation: Entry was made into the GDDM-REXX nucleus extension program while a GDDM-REXX request is still outstanding. This would occur, for example, if an ASREAD was issued, PA2 pressed to go into CMS SUBSET, and any GDDM-REXX request made.

ERX0111 S NUCXDROP of GDDM-REXX is about to require re-IPL of CMS
Explanation: The nucleus extension program has been entered on a service call from CMS NUCXDROP while a GDDM-REXX request is outstanding. This could occur, for example, if an ASREAD was issued, PA2 pressed to go into CMS SUBSET, and the CMS NUCXDROP command issued. Subsequent return from CMS SUBSET will find the support code for GDDM-REXX missing (thus causing a program check).

ERX0112 W Invalid language selection. Default assumed
Explanation: The language selection option was misspelled or is not one of the supported languages.

ERX0113 S Unable to locate language module. Command terminated
Explanation: The GDDMREXX TXTLIB file was either not available on any accessed disk, or the language module for the selected language was not found within the TXTLIB.

Listing of ERXMODEL EXEC

```
/*********************************************************************/
/*                                                                   */
/*  5664-336  GDDM-REXX                                              */
/*  (C) COPYRIGHT IBM CORP. 1987                                     */
/*  LICENSED MATERIALS - PROPERTY OF IBM                             */
/*                                                                   */
/* PROGRAM NAME:                                                     */
/*  GDDM-REXX Sample Program - ERXMODEL                              */
/*                                                                   */
/* DESCRIPTIVE NAME:                                                 */
/*  A template for constructing EXECs which will use GDDM-REXX.      */
/*                                                                   */
/* STATUS: VERSION 1 RELEASE 1                                       */
/*                                                                   */
/* FUNCTION:                                                         */
/*  Display file ERXMODEL ADMGDF with alphanumeric and graphic text */
/*  annotation; real purpose is to show how to code an EXEC which    */
/*  uses GDDM-REXX.  The prolog and epilog sections are recommended */
/*  in any EXEC which is to use GDDM-REXX; sub-procedure EXECs need */
/*  only code Address GDDM prior to issuing GDDM-REXX calls          */
/*                                                                   */
/* DEPENDENCIES:                                                     */
/*  Requires GDDM-REXX, GDDM/VM and                                  */
/*  files ERXMODEL ADMGDF, ADMUWKSF ADMSYMBL.                        */
/* RESTRICTIONS: None                                                */
/* ERROR MESSAGES:                                                   */
/*  'Unable to load GDDM-REXX'                                       */
/*  'Non-Zero return code from GDDM-REXX call: '                     */
/* ENTRY CONDITIONS:                                                 */
/*  No parameters required; '?' or 'HELP' will display prolog        */
/* EXIT CONDITIONS:                                                  */
/*  Exit with RC=99 for error conditions                             */
/* CHANGE ACTIVITY: None                                             */
/*                                                                   */
/*********************************************************************/
signal helpend
HELPEND: hend=sigl-1
Arg parm .

If parm='?' | parm='HELP' then Do    /*Display prolog comments      */
  Do i=1 to hend
    Say sourceline(i)
    End
  Exit
  End
```

```
/********************************************************************/
/*  Suggested prolog for all GDDM-REXX EXECs                       */
/*  External subroutine EXECs only require Address GDDM at start.  */
/********************************************************************/
/* 1. REXX requires a leading comment statement                    */

/* 2. Install GDDM-REXX (mainline EXECs only):                     */
Address command 'GDDMREXX INIT'
If rc<>0 then
   Do
      Say 'Unable to load GDDM-REXX'
      Exit 99
   End

/* 3. Turn REXX's attention to GDDM-REXX for subcommands           */
Address gddm
Signal on error  /* intercept any non-zero ret codes from GDDM-REXX */

/* 'FSINIT' */   /* optional, since initialization of GDDM-REXX    */
                 /* creates a default instance of GDDM.            */
/********************************************************************/
/*  End of prolog.  Begin your own GDDM-REXX application.          */
/********************************************************************/
/* sample application begins here.....replace with your own        */ |
Call NOSUBSET    /* ensure PA2 will be returned to us.             */ |
                                                                      |
'FSQDEV 4 (. . .Rows .Cols)' /* FIND SIZE OF SCREEN */               |
rows=rows-3                                                           |
cols=cols-1                                                           |
'GSFLD 1 1 .rows .cols'                                               | This
/* define some alphanumeric fields                                 */ |
'ASDFLD 1 'rows+1 '1 1 .cols 0'                                       |
'ASFCOL 1 1'                                                          | is
'ASDFLD 2 'rows+2 '1 1 .cols 0'                                       |
'ASFCOL 2 1'                                                          |
'ASDFLD 3 'rows+3 '1 1 .cols 0'                                       | the
'ASFCOL 3 1'                                                          |
'ASCPUT 1 . "                   GDDM-REXX Sample Program ERXMODEL"'   |
'ASCPUT 2 . "                 Welcome to the world of REXX with GDDM"' | part
'ASCPUT 3 . "                  GDDM-REXX 5664-336 (C) IBM Corp 1987"'  |
Call mysubprog   /* note: mysubprog must issue Address gddm if it  */ |
                 /*       is an external EXEC and is to use GDDM.  */ | to
/* display some nice graphic characters                            */ |
'GSLSS 2 ADMUWKSF 194'                                                |
'GSCS 194'                                                            | change
'GSCM 3'                                                              |
'GSCB 13 8'                                                           |
'GSCOL 6'                                                             |
'GSCHAR 7 85 . "GDDM-REXX"'                                           |
'ASREAD . . .'   /* force display and wait for any user action     */ |
/* etc...*/
/* end of sample application.... remove to here.                   */
```

```
/*******************************************************************/
/*  Epilog for all GDDM-REXX mainline EXECs:                       */
/*******************************************************************/
/* Then, at all exit points from the exec:                         */
/* 'FSTERM' */   /* optional, use only if 'FSINIT' issued above    */
Address command 'GDDMREXX TERM'  /* Terminate GDDM-REXX            */
Exit 0
/*******************************************************************/
/*  End of epilog.                                                 */
/*******************************************************************/

MYSUBPROG:  /* sample internal sub-procedure                       */
/* Address GDDM not required for internal procedures               */
/* load a GDF file into the current segment                        */
'GSLOAD ERXMODEL 2 (0 2) . 0 .'
Return

NOSUBSET:
/*******************************************************************/
/* Disables the GDDM default action on PA2.  Default               */
/* action is for GDDM to field PA2 and go into CMS SUBSET.         */
/* The following DSOPEN proc opts cause PA2 to be returned         */
/* to the application.                                             */
/*******************************************************************/
procopts.1=1000                      /* PA1/2 protocol             */
procopts.2=2                         /* get PA2 in application     */
'DSOPEN 9 1 * 2 .procopts. . ()'     /* Open device                */
'DSUSE 1 9'                          /* Use device                 */
Return

/*******************************************************************/
/* Error processing - non-zero return from GDDM-REXX call          */
/*******************************************************************/
ERROR:
Grc=rc
'GxGet Lastmsg .G_Msg'
Say 'Line:' sigl '-' sourceline(sigl)
Say 'Non-zero return code from GDDM-REXX call: ' Grc
Say G_Msg
Address command 'GDDMREXX TERM'  /* Terminate GDDM-REXX            */
Exit 99
```

Index

GDDM-REXX Guide

READER'S COMMENT FORM

Order No. SC33-0478-0

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Number of your latest Technical Newsletter for this publication . . .

Note: *Staples can cause problems with automated mail-sorting equipment. Please use pressure-sensitive or other gummed tape to seal this form.*

If you want an acknowledgement, give your name and address below.

Name . . . .

Job Title . . . . Company . . . .

Address. . . . .

. . . . Zip . . . .

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

SC33-0478-0

Reader's Comment Form

Fold and tape Please Do Not Staple Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 6R1H,
180 Kost Road,
Mechanicsburg, PA 17055, USA

Fold and tape Please Do Not Staple Fold and tape

IBM®

summary

Basic structure

`/* REXX comment */`	Must start with a comment
`Address command 'GDDMREXX INIT'`	Must initialize GDDM-REXX
`Address gddm`	Must address GDDM before any GDDM calls

`'GSLINE 50 50'`
`'GSCHAR 50 45 5 "Words"'`
	Quotes around GDDM calls to prevent REXX making alterations
	Double quotes around GDDM strings

`'ASREAD .type .val .count'`	Dots before variable names in GDDM calls

`If type=1 & val=2 then`
`   Call subprog`
	No dots before variable names in REXX statements

`Address command 'GDDMREXX TERM'`	Address commands and terminate GDDM-REXX at end
`Exit`	End EXEC with **Exit**

Array parameters

`onedimx.1=1; onedimx.2=2; ...`
`twodim.1.1=30; twodim.1.2=20; ...`
	Assign array values like this

`'GSPLNE 3 .onedimx. .onedimy.'`
`'GSPLNE 3 (1 2 3) (.top 5 .bot)'`
	Use array names like this or list names or values in parentheses

`'CHBAR 2 3 .twodim.'`
`'CHBAR 2 3 ((10 20 30) (5 11 17))'`
	For two dimensional arrays two sets of parentheses for list

The dot after the array name can be omitted in the GDDM call. Variable names then take the form **ARRAY1**, **ARRAY2** and so on. (Two-dimensional arrays are **ARRAY1.1**, **ARRAY1.2**).

Using dots for parameters

`'GSCHAR 50 50 . "character string"'`	You can replace deducible values with dots; the length of string you pass is omitted
`'ASCPUT 1 . .string'`	... and again
`'ASCGET 1 .length .string'`	... but NOT when string is returned
`'ASREAD . . .'`	Also returned variables of no interest, values from **ASREAD** will be ignored

Syntax of GDDM-REXX commands, subcommands, and utility EXEC	*Page*
GDDMREXX INIT [([NODCSS] [LANG x] [)]]	63
GDDMREXX TERM [(ALL[)]]	63
GDDMREXX VERSION [(STACK|LIFO|FIFO[)]]	63
GXGET AAB .aabtoken	64
GXGET CDT .name .entry	64
GXGET LASTMSG .msg	64
GXGET MSG .state .level	64
GXGET NAMES .namelist	65
GXGET TRACE .state .time	65
GXSET AAB .aabtoken	65
GXSET MSADS .mapgrp .map .prefix .ads	65
GXSET MSG .state [.level]	66
GXSET MSVARS .mapgrp .map .prefix .ads	66
GXSET TRACE .state [.time]	66
ERXMSVAR mapgroupname mapname prefix	66

SC33-0478-0
File No. S370/4300/CMS-34

GDDM-REXX Guide Printed in USA SC33-0478-0

SC33-0478-00

IBM®

102656637