Cco - Module
Owner's Manual
For the HP-41
Foreword

The CCD-Module was developed with the goal of providing a tool for improving all applications of the HP-41, and to simplify programming. User friendliness was an objective of particular importance in obtaining this goal.

The CCD-Module should support the programmer to such an extent that he can concentrate on the actual problem he is attempting to solve, rather than expend a great deal of effort in the mechanics of programming. Other sections of the program such as the formatting of input and output, can be assembled with the new functions supplied by the module, and problems which were previously insoluble can be solved.

More than a year was required for the development of this module. Special thanks are due for the support furnished by all of members of CCD (Computer Club of Germany), who made possible the production of this module in its present form, by their strong interest, by proposing routines for functions to be included in the module, and not least by the patience they exhibited. Further thanks are due to Dr. Baltes, the spiritual father of the module, through whose work in coordinating the programming much room was created for new functions; also to Mr. Holger Adelmann by whose stimulus and programming efforts the CCD-Module was optimized. Furthermore I thank everyone who tested our module during its development phase, and who enhanced the success of this handbook by contributing applications for individual function. In this regard I am particularly grateful to Mr. Gerhard Kruse, who has written a variety of excellent and optimized programs for the handbook, and to Mr. Andreas Meyer for his literary work and for his superb preparation of figures. Furthermore I thank the Hewlett-Packard Company in Germany for their support, and last but not least, my friends Ken Emery, Stephan Abramson and Jeremy Smith who by their assistance in translation, have helped to make this module known throughout the world.

W&W Software Products GmbH

Wilfried Kötz, President

(C) Copyright W&W Software Products GmbH 1985
Chapter Index

1 Internal Design of the HP-41
2 Operating System Enhancements
3 Functions from Catalog 2 (-W&W FNS)
4 Matrix Functions (-ARR FNS)
5 Binary Functions (-HEX FNS)
6 Input/Output Functions (-I/O FNS)
7 Functions for Advanced Programming (-ADV FNS)
8 XF/Memory Functions (-XF/M FNS)
9 Bar Codes
10 Function Index
11 Compatibility
12 Literature
Subject Index

<table>
<thead>
<tr>
<th>1CMP</th>
<th>5.19</th>
</tr>
</thead>
<tbody>
<tr>
<td>1's Complement Mode</td>
<td>5.12</td>
</tr>
<tr>
<td>2's Complement Mode</td>
<td>5.12</td>
</tr>
<tr>
<td>2CMP</td>
<td>5.20</td>
</tr>
<tr>
<td>&gt;C+</td>
<td>4.19</td>
</tr>
<tr>
<td>&gt;R+</td>
<td>4.21</td>
</tr>
<tr>
<td>&quot;?&gt;CAS&quot;</td>
<td>6.15</td>
</tr>
<tr>
<td>?IJ</td>
<td>4.14</td>
</tr>
<tr>
<td>?IJA</td>
<td>4.15</td>
</tr>
<tr>
<td>&quot;A?&quot; (Bar Code on Page 9 .05)</td>
<td>7.25</td>
</tr>
<tr>
<td>&quot;ABIN&quot; (Bar Code on Page 9 .05)</td>
<td>4.77</td>
</tr>
<tr>
<td>ABSP</td>
<td>6.29</td>
</tr>
<tr>
<td>ACAXY</td>
<td>6.17</td>
</tr>
<tr>
<td>ACLX</td>
<td>6.22</td>
</tr>
<tr>
<td>Address Structure of the HP-41 Status Registers</td>
<td>2.14</td>
</tr>
<tr>
<td>ALPHA Functions</td>
<td>6.29</td>
</tr>
<tr>
<td>AND</td>
<td>5.27</td>
</tr>
<tr>
<td>ARCL</td>
<td>2.13</td>
</tr>
<tr>
<td>ARCLE</td>
<td>6.32</td>
</tr>
<tr>
<td>ARCLH</td>
<td>6.35,5.25</td>
</tr>
<tr>
<td>ARCLI</td>
<td>6.37</td>
</tr>
<tr>
<td>ASN</td>
<td>2.11</td>
</tr>
<tr>
<td>ASTO</td>
<td>2.13</td>
</tr>
<tr>
<td>bC?</td>
<td>5.43</td>
</tr>
<tr>
<td>bS?</td>
<td>5.41</td>
</tr>
<tr>
<td>&quot;BS?&quot; (Bar Code on Page 9 .06)</td>
<td>8.08</td>
</tr>
<tr>
<td>B?</td>
<td>3.05</td>
</tr>
<tr>
<td>Bar Codes for the Functions of the CCD-Module</td>
<td>9.18</td>
</tr>
<tr>
<td>Bar Codes for Programs</td>
<td>9.05</td>
</tr>
<tr>
<td>Bar Codes for the Functions of the CCD-Module</td>
<td>9.18</td>
</tr>
<tr>
<td>Basic Setup (-HEX FNS)</td>
<td>5.16</td>
</tr>
<tr>
<td>Binary Number System</td>
<td>5.05</td>
</tr>
<tr>
<td>Byte/Function Table</td>
<td>1.20</td>
</tr>
<tr>
<td>C&lt;&lt;C</td>
<td>4.31</td>
</tr>
<tr>
<td>C&gt;+.</td>
<td>4.23</td>
</tr>
<tr>
<td>C&gt;-</td>
<td>4.25</td>
</tr>
<tr>
<td>Calculating of Absolute Adresses</td>
<td>7.09</td>
</tr>
<tr>
<td>CAS</td>
<td>3.09</td>
</tr>
<tr>
<td>Catalogs</td>
<td>2.05</td>
</tr>
<tr>
<td>Cb</td>
<td>5.45</td>
</tr>
<tr>
<td>&quot;CB&quot; (Bar Code on Page 9 .07)</td>
<td>7.36</td>
</tr>
<tr>
<td>&quot;CDE&quot; (Bar Code on Page 9 .07)</td>
<td>7.15</td>
</tr>
</tbody>
</table>

(C) Copyright W&W Software Products GmbH 1985
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;CF55&quot; (Bar Code on Page 9.08)</td>
<td>5.50</td>
</tr>
<tr>
<td>&quot;CHK&quot; (Bar Code on Page 9.08)</td>
<td>7.34</td>
</tr>
<tr>
<td>CLA-</td>
<td>6.30</td>
</tr>
<tr>
<td>CLB</td>
<td>3.07</td>
</tr>
<tr>
<td>&quot;CLK&quot; (Bar Code on Page 9.08)</td>
<td>8.11</td>
</tr>
<tr>
<td>CMAXAB</td>
<td>4.43</td>
</tr>
<tr>
<td>CNRM</td>
<td>4.54</td>
</tr>
<tr>
<td>Complement</td>
<td>5.10</td>
</tr>
<tr>
<td>Complement (Signed) Modes</td>
<td>5.11</td>
</tr>
<tr>
<td>Complement Notation</td>
<td>5.13</td>
</tr>
<tr>
<td>Construction and Manipulation of Arrays</td>
<td>4.05</td>
</tr>
<tr>
<td>Construction of Arrays</td>
<td>4.06</td>
</tr>
<tr>
<td>CSUM</td>
<td>4.52</td>
</tr>
<tr>
<td>Data Registers</td>
<td>1.11</td>
</tr>
<tr>
<td>DCD</td>
<td>7.14</td>
</tr>
<tr>
<td>Decoding Function</td>
<td>7.14</td>
</tr>
<tr>
<td>Determining the Extreme Values of Array Elements</td>
<td>4.39</td>
</tr>
<tr>
<td>DIM</td>
<td>4.13</td>
</tr>
<tr>
<td>Direct and Indirect Memory Access Functions</td>
<td>2.13</td>
</tr>
<tr>
<td>Display Format Instructions</td>
<td>1.23</td>
</tr>
<tr>
<td>Distribution of Ranges of Values</td>
<td>5.11</td>
</tr>
<tr>
<td>END Instruction</td>
<td>1.16</td>
</tr>
<tr>
<td>END Instructions</td>
<td>1.25</td>
</tr>
<tr>
<td>Examples of the Contents in a Numeric Register</td>
<td>1.14</td>
</tr>
<tr>
<td>F/E</td>
<td>6.27</td>
</tr>
<tr>
<td>FIX/ENG Mode</td>
<td>6.27</td>
</tr>
<tr>
<td>Flag and Display Format Instructions</td>
<td>1.23</td>
</tr>
<tr>
<td>FNRM</td>
<td>4.56</td>
</tr>
<tr>
<td>Functions for Calculating of Absolute Addresses</td>
<td>7.09</td>
</tr>
<tr>
<td>Functions for the Construction and Manipulation of Arrays</td>
<td>4.05</td>
</tr>
<tr>
<td>Functions for Determining the Extreme Values of Array Elements</td>
<td>4.39</td>
</tr>
<tr>
<td>Functions for Manipulating the Program Pointer</td>
<td>7.16</td>
</tr>
<tr>
<td>Functions for Printer Output</td>
<td>6.17</td>
</tr>
<tr>
<td>Functions for Shifting and Exchanging Elements</td>
<td>4.31</td>
</tr>
<tr>
<td>Functions for the Manipulation of Element Pointers</td>
<td>4.14</td>
</tr>
<tr>
<td>Functions for the Manipulation of Individual Bits</td>
<td>5.33</td>
</tr>
<tr>
<td>&quot;GE&quot; (Bar Code on Page 9.09)</td>
<td>7.27</td>
</tr>
<tr>
<td>GETB</td>
<td>8.07</td>
</tr>
<tr>
<td>GETK</td>
<td>8.10</td>
</tr>
<tr>
<td>GTO and XEQ Instructions</td>
<td>1.25</td>
</tr>
<tr>
<td>Hexadecimal Byte/Function Table</td>
<td>1.20</td>
</tr>
<tr>
<td>Hexadecimal Number System</td>
<td>5.09</td>
</tr>
<tr>
<td>&quot;H-O&quot; (Bar Code on Page 9.09)</td>
<td>6.11</td>
</tr>
<tr>
<td>I/O Buffers (Design and Inner Structure)</td>
<td>1.33</td>
</tr>
<tr>
<td>IJ=</td>
<td>4.16</td>
</tr>
<tr>
<td>IJ=A</td>
<td>4.17</td>
</tr>
</tbody>
</table>

(C) Copyright W&W Software Products GmbH 1985
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPT</td>
<td>6.05</td>
</tr>
<tr>
<td>Input and Output Functions for Data Arrays</td>
<td>4.18</td>
</tr>
<tr>
<td>Input and Output Functions for Use with -HEX FNS</td>
<td>5.22</td>
</tr>
<tr>
<td>Input Functions</td>
<td>6.05</td>
</tr>
<tr>
<td>Input of Any Alpha Character String</td>
<td>2.15</td>
</tr>
<tr>
<td>&quot;INV&quot; (Bar Code on Page 9.10)</td>
<td>4.83</td>
</tr>
<tr>
<td>&quot;KEY&quot;</td>
<td>6.14</td>
</tr>
<tr>
<td>Key Assignments (Design and Inner Structure)</td>
<td>1.31</td>
</tr>
<tr>
<td>Label Instructions</td>
<td>1.23</td>
</tr>
<tr>
<td>Layout of a Numeric Register</td>
<td>1.13</td>
</tr>
<tr>
<td>Line Numbering</td>
<td>1.30</td>
</tr>
<tr>
<td>M+</td>
<td>4.57</td>
</tr>
<tr>
<td>M-</td>
<td>4.59</td>
</tr>
<tr>
<td>M*</td>
<td>4.61</td>
</tr>
<tr>
<td>M*M</td>
<td>4.65</td>
</tr>
<tr>
<td>M/</td>
<td>4.63</td>
</tr>
<tr>
<td>Main Memory</td>
<td>1.09</td>
</tr>
<tr>
<td>Main Memory Configuration</td>
<td>1.10</td>
</tr>
<tr>
<td>Manipulating the Program Pointer</td>
<td>7.16</td>
</tr>
<tr>
<td>Manipulation of Element Pointers</td>
<td>4.14</td>
</tr>
<tr>
<td>Manipulation of Individual Bits</td>
<td>5.33</td>
</tr>
<tr>
<td>MAX</td>
<td>4.39</td>
</tr>
<tr>
<td>MAXAB</td>
<td>4.41</td>
</tr>
<tr>
<td>Memory Access Functions</td>
<td>2.13</td>
</tr>
<tr>
<td>MDIM</td>
<td>4.09</td>
</tr>
<tr>
<td>MIN</td>
<td>4.45</td>
</tr>
<tr>
<td>Modes</td>
<td>5.11</td>
</tr>
<tr>
<td>MOVE</td>
<td>4.35</td>
</tr>
<tr>
<td>MRGK</td>
<td>8.12</td>
</tr>
<tr>
<td>Negative Numbers</td>
<td>5.10</td>
</tr>
<tr>
<td>NOT</td>
<td>5.32</td>
</tr>
<tr>
<td>Number Input</td>
<td>1.28</td>
</tr>
<tr>
<td>Number Systems</td>
<td>5.05</td>
</tr>
<tr>
<td>Octal Number System</td>
<td>5.07</td>
</tr>
<tr>
<td>One Byte Instructions</td>
<td>1.22</td>
</tr>
<tr>
<td>OR</td>
<td>5.29</td>
</tr>
<tr>
<td>Organization and Construction of Arrays</td>
<td>4.06</td>
</tr>
<tr>
<td>Output Functions</td>
<td>6.17</td>
</tr>
<tr>
<td>Output Functions for Data Arrays</td>
<td>4.18</td>
</tr>
<tr>
<td>Output Functions for Use with -HEX FNS</td>
<td>5.22</td>
</tr>
<tr>
<td>&quot;PBC&quot; (Bar Code on Page 9.11)</td>
<td>7.40</td>
</tr>
<tr>
<td>PC&lt;&gt;RTN</td>
<td>7.21</td>
</tr>
<tr>
<td>PC&gt;X</td>
<td>7.16</td>
</tr>
<tr>
<td>PEEK and POKE Functions</td>
<td>7.24</td>
</tr>
<tr>
<td>PEEKB</td>
<td>7.24</td>
</tr>
<tr>
<td>PEEKR</td>
<td>7.29</td>
</tr>
</tbody>
</table>
PHINPT" (Bar Code on Page 9.12) ............................ 6 .07
Physical Address Structure of the HP-41 Status

Registers .................................................. 2 .14
PIV .......................................................... 4 .46
PHD .......................................................... 7 .08
"PK" (Bar Code on Page 9 .12) ......................... 8 .13
PLNG ......................................................... 7 .05
PMTA .......................................................... 6 .09
PMTH ......................................................... 6 .11;5 .22
PMTK .......................................................... 6 .13
POKE Functions .............................................. 7 .24
POKEB ......................................................... 7 .31
POKER ......................................................... 7 .35
PPLNG ......................................................... 7 .07
"PR" ............................................................ 6 .18
"PR1" (Bar Code on Page 9 .13) ....................... 6 .22
"PR4" (Bar Code on Page 9 .14) ....................... 6 .24
PRAXY .......................................................... 6 .20
Printer Output ................................................ 6 .17
PRL ............................................................. 6 .24
Program Code of the HP-41 ........................... 1 .19
Program Line Numbering ................................. 1 .30
Program Memory (Design) .............................. 1 .15
Representation of Negative Numbers ...................... 5 .10
R-PR .......................................................... 4 .73
R-QR .......................................................... 4 .69
R< ............................................................. 5 .37
R<>R ........................................................... 4 .33
R> ............................................................. 5 .39
R>>+ ........................................................... 4 .27
R>- ............................................................ 4 .29
R>R? .......................................................... 4 .47
RCL ............................................................. 2 .13
Register Functions .......................................... 1 .22
RMAXAB ......................................................... 4 .44
RNDM .......................................................... 3 .10
RNRM .......................................................... 4 .55
ROM Memory Configuration ............................ 2 .10
RSUM ........................................................... 4 .53
S< ............................................................. 5 .33
S> ............................................................. 5 .35
SAS ............................................................ 3 .08
SAVEB .......................................................... 8 .05
SAVEK .......................................................... 8 .09
Sb .............................................................. 5 .47
SEED .......................................................... 3 .12
Setting FIX/ENG Mode ..................................... 6 .27
SORT ............................................................ 3 .13

(C) Copyright W&W Software Products GmbH 1985
Introduction

The CCD-Module is an extension for the HP-41 handheld calculator, which expands the calculator's vocabulary by nearly a 100 new functions. The module also provides several enhancements to the operating system of the HP-41, such as new catalogs and the capability of direct keyboard entry of lower case letters.

The ideas and wishes of many members of CCD (Computer Club of Germany) were particularly taken into account in planning the module; thus the module is named after the group.

The CCD-Module was programmed in machine code (MCODE), which sets it apart from other application modules. This design approach affords accuracy and rapid execution of the functions of the CCD-Module, which would have otherwise been unattainable.

The CCD-Module is a valuable addition to a "bare-bones" (i.e. an otherwise unenhanced) HP-41; however, as the size of the system increases, so do the possibilities afforded by the module. Thus, in particular, the CCD-Module supports all printer characters, the Extended Functions/Memory module and the Hewlett-Packard Interface Loop (HP-IL).
Chapter 1

Internal Design of the HP-41
## Contents Chapter 1

### Internal Design of the HP-41

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal Design of the HP-41</td>
<td>1.05</td>
</tr>
<tr>
<td>The RAM of the HP-41</td>
<td>1.06</td>
</tr>
<tr>
<td>The Structure of RAM</td>
<td>1.07</td>
</tr>
<tr>
<td>Main Memory</td>
<td>1.09</td>
</tr>
<tr>
<td>Main Memory Configuration</td>
<td>1.10</td>
</tr>
<tr>
<td>Data Registers</td>
<td>1.11</td>
</tr>
<tr>
<td>Data Registers (Inner Structure)</td>
<td>1.13</td>
</tr>
<tr>
<td>The Register Structure</td>
<td>1.13</td>
</tr>
<tr>
<td>Layout of a Numeric Register</td>
<td>1.13</td>
</tr>
<tr>
<td>Examples of the Contents in a Numeric Register</td>
<td>1.14</td>
</tr>
<tr>
<td>Design of an ASCII Register</td>
<td>1.14</td>
</tr>
<tr>
<td>Examples for the Contents of an ASCII Register</td>
<td>1.15</td>
</tr>
<tr>
<td>Program Memory (Design)</td>
<td>1.15</td>
</tr>
<tr>
<td>The Meaning of the END Instruction</td>
<td>1.16</td>
</tr>
<tr>
<td>Program Memory (Inner Structure)</td>
<td>1.17</td>
</tr>
<tr>
<td>How a Program Consists of Instructions</td>
<td>1.17</td>
</tr>
<tr>
<td>The Program Code of the HP-41</td>
<td>1.19</td>
</tr>
<tr>
<td>Hexadecimal Byte/Function Table</td>
<td>1.20</td>
</tr>
<tr>
<td>One Byte Instructions</td>
<td>1.22</td>
</tr>
<tr>
<td>Flag and Display Format Instructions</td>
<td>1.23</td>
</tr>
<tr>
<td>Label Instructions</td>
<td>1.23</td>
</tr>
<tr>
<td>END Instructions</td>
<td>1.25</td>
</tr>
<tr>
<td>GTO and XEQ Instructions</td>
<td>1.25</td>
</tr>
<tr>
<td>Text Instructions</td>
<td>1.27</td>
</tr>
<tr>
<td>Number Input</td>
<td>1.28</td>
</tr>
<tr>
<td>Instructions from Plug In Modules</td>
<td>1.28</td>
</tr>
<tr>
<td>Summary</td>
<td>1.30</td>
</tr>
<tr>
<td>Program Line Numbering</td>
<td>1.30</td>
</tr>
<tr>
<td>Instructions in Plug In Modules</td>
<td>1.31</td>
</tr>
<tr>
<td>Key Assignments (Design and Inner Structure)</td>
<td>1.31</td>
</tr>
<tr>
<td>I/O Buffers (Design and Inner Structure)</td>
<td>1.33</td>
</tr>
</tbody>
</table>
Internal Design of the HP-41

Many functions of the CCD-Module (PEEK, POKE and "Synthetic") operate on all of the HP-41's RAM. This RAM includes the main memory (data registers, programs, key assignments and I/O buffers), the status registers (stack, flag register and other informations for the operating system) and extendend memory. The operating system uses memory locations to store status information. The careless use of some registers can crash your HP-41, lose the memory contents but will never harm the machine itself.

First, we will explain the design and the organization of RAM.

The HP 41 has two different types of memory

- Random Access Memory (RAM)
- Read Only Memory (ROM)

As only RAM can be altered by the user, its design and logical organization is the most interesting for us and will be covered first.
The RAM of the HP-41

HP-41 RAM comprises space for 1024 registers which are each 56 bits wide. The CPU is only capable of reading one register at a time. If it wants to alter one bit, it has to read the whole register, set or clear the bit and then write the whole register back again. Inside the CPU the register contents can be altered all together or in parts. The smallest part is the bit. Other parts are a nybble (4 bits) or a byte (8 bits). Bits, nybbles and bytes in a register are numbered starting with 0 from right to left. The bits are counted from 0 to 55, the nybbles from 0 to 13, and the bytes from 0 to 6.

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>Byte number</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>55...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

We should not not forget that this is only a logical division in a register.

The HP-41 CPU is able to address 1024 registers. The registers are numbered from 0 to 1023. The register number is an absolute value. It is only dependent on the physical location of the register in RAM. That the CPU is able to address 1024 registers does not mean that they are available to the user because the operating system needs certain gaps for its own use.
The Structure of RAM

HP-41 Memory Configuration

Up to now we only talked about the physical configuration of RAM. We shall now discuss the logical configuration of RAM. RAM is subdivided into three parts, according to its use:

1) status registers
2) main memory
3) extended memory
<table>
<thead>
<tr>
<th>Designation</th>
<th>Start Address - End Address (\text{decimal})</th>
<th>Start Address - End Address (\text{hexadecimal})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Registers</td>
<td>0 — 15</td>
<td>000 — 00F</td>
</tr>
<tr>
<td>Main Memory</td>
<td>192 — 511</td>
<td>0C0 — 1FF</td>
</tr>
<tr>
<td>Extended Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X Function Mem.</td>
<td>64 — 191</td>
<td>040 — 0BF</td>
</tr>
<tr>
<td>X Memory 1</td>
<td>513 — 751</td>
<td>201 — 2EF</td>
</tr>
<tr>
<td>X Memory 2</td>
<td>769 — 1007</td>
<td>301 — 3EF</td>
</tr>
</tbody>
</table>

We recognize that the extended memory is split in 3 parts (extendend functions and 2 extended memory). In the HP-41 C main memory ranges from register 192 to 255. It can be expanded in 64 register increments till the limit of 511 is reached.
Main Memory

Four different kinds of data can be stored in the main memory:

- alphanumeric data
- programs
- key assignments
- I/O buffers

There is a special place in memory reserved for each data type. The operating system allocates space for each type of data on request.
Main Memory Configuration:

The operating system has four different modes of data storage:

- data storage
- program storage
- KA-storage
- I/O Buffer storage

Free memory is allocated for use by the operating system.
Data Registers

The data registers are located in the upper part of main memory. The commands STO and ASTO are used to save all kinds of numeric and alphanumeric data in the data registers. The register numbers used for that will be called "relative numbers" in this book.

The upper boundary for data registers is 511 absolute in the CV and CX, but in the C it is dependent on the number of memory modules plugged in.

<table>
<thead>
<tr>
<th>Memory modules</th>
<th>Address (decimal)</th>
<th>Address (hexadecimal)</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>192 – 255</td>
<td>0CO – 0FF</td>
<td>HP-41 C basic configuration</td>
</tr>
<tr>
<td>1</td>
<td>256 – 319</td>
<td>100 – 13F</td>
<td>or Quad RAM; note that for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>140 – 17F</td>
<td>the HP-41 CV and HP-41 CX</td>
</tr>
<tr>
<td>2</td>
<td>320 – 383</td>
<td>180 – 1BF</td>
<td>all of these addresses are</td>
</tr>
<tr>
<td>3</td>
<td>384 – 447</td>
<td>1CO – 1FF</td>
<td>permanently built in.</td>
</tr>
<tr>
<td>4</td>
<td>448 – 511</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The lower boundary is assigned by the operating system after executing the SIZE function.

Lower boundary = upper boundary + 1 - SIZE
As the upper boundary is 511 in most cases the formula reduces to:

lower boundary = 512 - SIZE

The lower boundary is called the curtain because it separates data registers from programs. In numbering the data registers, which is assigning relative addresses to the data registers, one starts at the curtain with 00.

The relation between absolute and relative addresses is shown by the table below:

<table>
<thead>
<tr>
<th>Absolute Address</th>
<th>Relative Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper boundary</td>
<td>Size-1</td>
</tr>
<tr>
<td>Curtain</td>
<td>0</td>
</tr>
<tr>
<td>Curtain + nnn</td>
<td>Register nnn</td>
</tr>
</tbody>
</table>

One can imagine that the operating system needs the curtain to change relative into absolute addresses. Because of this the address of the curtain is stored in one of the status registers. The curtain is only a logical division between data and program registers.

Having explained where the data registers are, and how they are counted, numbered and addressed we will now explain how the data is actually stored in a register.
Data Registers (Inner Structure)

An important rule for this:

1) Everything we store with STO or ASTO occupies a whole register
2) There are no physical boundaries, only logical boundaries

The Register Structure

We distinguish between numeric data (numbers) and alphanumeric or ASCII data.

Layout of a Numeric Register
Examples of the Contents in a Numeric Register

\[
\begin{array}{cccccccccccc}
9 & 8 & 7 & 5 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\Rightarrow -8,7512
\]

\[
\begin{array}{cccccccccccc}
0 & 8 & 7 & 5 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\Rightarrow 8,7512
\]

\[
\begin{array}{cccccccccccc}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1,2 \\
\end{array}
\Rightarrow 1,0000 \ 12
\]

\[
\begin{array}{cccccccccccc}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\Rightarrow 1,0000
\]

\[
\begin{array}{cccccccccccc}
9 & 8 & 7 & 5 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 9 & 8 & 5 \\
\end{array}
\Rightarrow -8,7512 -15
\]

The digit 0 indicates a positive number and the digit 9 indicates a negative number. The system in which each nybble contains a number between 0 and 9 is called BCD (Binary Coded Decimal). This means that nybble 13 of a numeric data register must either contain a 0 or a 9.

Design of an ASCII Register

\[
\begin{array}{cccccccccccc}
1 & X & & & & & & & & & & & & \\
\end{array}
\]

All ASCII registers have the digit 1 in nybble 13. In the bytes 0 to 5 there can be up to 6 ASCII letters right justified. If a register contains only one letter the bytes 1 to 5 are null bytes and byte 0 contains the letter.
Examples for the Contents of an ASCII Register:

\[
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 4 & 1 & 4 & 2 & 4 & 3 & 2 & 0 \\
\end{array}
\] \equiv "A B C"

\[
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 4 & 1 & 4 & 2 & 4 & 3 \\
\end{array}
\] \equiv "A B C"

\[
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 1 & 4 & 2 & 4 & 3 \\
\end{array}
\] \equiv "A B C"

\[
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 0 & 2 & 0 & 4 & 1 & 4 & 2 & 4 & 3 & 2 & 0 \\
\end{array}
\] \equiv "A B C"

It is simple to determine the difference between number and ASCII registers: one must only analyse nybble 13. If it is 1 it is an ASCII register, if it is 0 or 9 it is a data register. The nybble 12 can have any value in calculators manufactured with serial numbers less than 2036.... All machines manufactured since then place a zero in nybble 12.

The CCD-Module contains a function which makes it very simple to analyse a register: \textbf{DCD} (DeCoDe the X register).

**Program Memory (Design)**

Program memory starts with the register below the curtain (see the main memory map). It ends with the \textbf{.END.} instruction. The structure of the program registers is exactly the opposite of the data registers, the first program is in the topmost register and later programs are in lower numbered registers.

(C) Copyright W&W Software Products GmbH 1985 1.15
The Meaning of the **END** Instruction

The significance of the **END** instruction. The beginning of a program is either behind the **END** of the last program or directly below the curtain. Every **END** instruction contains information for how far away is the next highest **LBL** or **END**. The shortest possible program is only an **END**.

When we start with a "MEMORY LOST" and key in some program steps then we have a program with an **END** already attached. That is easy to see: just key RTN and R/S (in run mode). The operating system has a permanent **END**: the **.END.** instruction. This means that after "MEMORY LOST" there already exists a program in memory: the **.END.**

The **.END.** has a special function: it is the last **END** in program memory and can not be deleted by normal means, nor should it be deleted by any other means.

When one writes or debugs programs and new steps are written into memory, null bytes are overwritten. When there are no null bytes available the operating system produces some. To do this the operating system shifts program memory down 1 register into the free memory space. By doing that 7 null bytes are produced. The information about the program length in the **END** is renewed.

"PACKING" removes unnecessary null bytes from program memory. Here too, the length information in the **END** is renewed.
Program Memory (Inner Structure)

How a Program Consists of Instructions

The operating system of the HP-41 has a list of all functions (CAT 3) each having a special code. When a function is keyed in (with XEQ ALPHA function ALPHA), the operating system searches this list. When the function is found the operating system stores the code and branches (if necessary) to the prompt for the argument byte. Then the function code and argument can be saved as program step.

A special feature of the HP-41 is that the function list can be increased by plug-in modules (such as the CCD-Module) and with programs created by the user.

The function list is made visible with the CAT instruction.

When the operating system searches for a program or a function it does so in the following sequence:

- user programs in the reverse order than they appear in catalog 1
- instructions from plug in modules starting with page 5 (TIME module)
- internal function list (CAT 3)
After keying in a function the operating system first makes a syntax check, making it normally impossible for us to key in functions like STO M or TONE 57. After the syntax check the operating system stores the function and argument code in the memory.

While executing a program the operating system reads the program line by line and interprets each instruction by branching to the appropriate machine code program. (There is a machine code program for every instruction). This part of the operating system is called the interpreter.

The interpreter works without knowing how the program code was assembled.

An understanding of this is the basics of synthetic programming. Synthetic programming is nothing more than making a program code by bypassing the syntax check.

Up to here we have only explained how an instruction is produced in program memory via the syntax check. We will now explain what the instruction itself looks like.
The Program Code of the HP-41

The program code of the HP-41 is byte oriented. This means that the interpreter works byte by byte through program memory. It works from top to bottom, or from high addresses to low.

256 instructions can be coded with one byte. However, the number of instructions alone from STO, RCL and ASTO with register numbers 00-99 is greater than 256. This problem is avoided in the HP-41 by using an argument byte (the second byte is an argument byte for an STO instruction). Depending on the number of argument bytes used we may have one, two, three or multiple byte instructions.

In all program lines the first byte is the instruction and all following bytes are the arguments.

In the following byte table all possible instructions are shown. The table is shown as a $16 \times 16$ matrix. This is because every byte can be sliced into two nybbles, the first nybble is the row number the second is the column number. Instructions in one row are always of the same type.

On the next two pages you will see the HEX table for the HP-41. Every square is structured like the following diagram:
## Hexadecimal Byte/Function Table

### HP-41C Quick Reference Card for Synthetic Programming © 1982, Synthetix

<table>
<thead>
<tr>
<th>REGISTER</th>
<th>1/0</th>
<th>1/2</th>
<th>1/3</th>
<th>4/5</th>
<th>6/7</th>
<th>8/9</th>
<th>A/B</th>
<th>C/D</th>
<th>E/F</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARRY</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>STATUS</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>RESULT</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>INDEX</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>INVERSE</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>INDEX</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>INVERSE</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

### FLAGS (Register d)

- 33 IL absolute manual
- 34 not used purpose
- 35 not used
- 36 auto execute 39 number
- 37 doublewide of digits
- 38 lower case 41 display
- 39 overwrite 0 0 SCI
- 40 16 IL printer 0 1 ENG
- 41 0 MAN 1 0 FIX
- 42 0 1 NORM 1 1 FIX/ENG
- 43 0 1 TRACE 42-43 trig mode
- 44 1 1 TR/STACK 0 0 DEG
- 45 record 0 1 RAD
- 46 incomplete 1 0 GRAD
- 47 general use 1 1 RAD
- 48 cleared at 44 cont. ON
- 49 0 turn-on 45 system
- 50 0 prtr enable 21 data entry
- 51 0 num. entry 46 partial key
- 52 0 alpha entry 47 sequence
- 53 0 range ignore 52 0 
- 54 0 error ignore 47 SHIFT
- 55 0 audio enable 48 0 low BAT
- 56 0 USER mode 50 0 message
- 57 0 dec./comma 51 0 SST
- 58 0 digit grouping 52 0 PGM
- 59 0 CAT 53 0 I/O
- 60 0 timer 54 0 PSE
- 61 0 DMY/MDY 55 0 printer
- 62 0 manual 56 0 existence

### Bit Numbers in a 7-byte Register

- 0000
- 0001
- 0010
- 0011
- 0100
- 0101
- 0110
- 0111
- 1000
- 1001
- 1010
- 1011
- 1100
- 1101
- 1110
- 1111

---

(C) Copyright W&W Software Products GmbH 1985
**HP-41C QUICK REFERENCE CARD FOR SYNTHETIC PROGRAMMING © 1982, SYNTHETIX**

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>128 F</td>
<td>STA</td>
<td>IND</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>RCL</td>
<td>IND</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>A</td>
<td>XR</td>
<td>0</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>B</td>
<td>SPARE</td>
<td>48</td>
<td>47</td>
<td>16</td>
</tr>
<tr>
<td>C</td>
<td>GLOBAL</td>
<td>64</td>
<td>63</td>
<td>16</td>
</tr>
<tr>
<td>D</td>
<td>GTO</td>
<td>IND</td>
<td>80</td>
<td>81</td>
</tr>
<tr>
<td>E</td>
<td>IMAGE</td>
<td>96</td>
<td>95</td>
<td>16</td>
</tr>
<tr>
<td>F</td>
<td>TEXT</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

### Structure of multi-byte instructions

**Two-byte instructions**

- **STO**: STORE
- **SIN**: SINUS
- **TAN**: TANGENT
- **LOG**: LOGARITHM
- **10**: DECIMAL
- **π**: PI
- **ABS**: ABSOLUTE VALUE
- **BASE**: BASE

**Two-byte special cases**

- **GTO**: GOTO
- **XEQ**: XEQ
- **IND**: IND
- **SPARE**: SPARE
- **XROM**: XROM
- **GTO**
- **IND**

**Short form GTO**

- **GTO**
- **IND**

**Three-byte instructions**

- **IND**, **IND**, **IND**
- **IND**, **IND**, **IND**

**Variable length instructions**

- **TEXT**: TEXT
- **APPEND**: APPEND
- **CHAR**: CHAR

**Internal Design**

For price information and a list of dealers in your area, send a self-addressed stamped envelope to SYNTHETIX, 1540 Mathews Ave., Manhattan Beach, CA 90266, USA.
One Byte Instructions

Without exception all instructions in row 0 and 2 through 8 are one byte instructions.

Register Functions

The functions ARCL, ASTO, DSE, ISG, RCL, }REG, ST+, ST-, ST*, ST/ and X<> are register instructions. All expect a register number as an argument. These instructions are two byte instructions. The first byte specifies the action, the second the register number (coded hexadecimally). The codes are hex 00 to hex 63 for the registers numbered 0-99 and hex 70 to hex 74 are used for the stack registers T, Z, Y, X and L.

The instruction RCL 87 is coded as follows: Hex 90 57

The first byte (instruction Byte) is the RCL, the second (argument byte) is 87.

What happens when there is a RCL IND 87 in the program memory?

When a register is used indirectly, hex 80 is added to the argument byte, that means hex 57 + hex 80 = D7. You can see the different arguments and their hex equivalents in the byte Table.

There is one special thing with RCL and STO Instructions: As they are very often used with register numbers up to 15 there are special one byte instructions used for this purpose (rows 2 and 3). This not only helps to save memory space but even speeds up instruction time.
Flag and Display Format Instructions

The functions CF, FC?, FC?C, FS?, FS?C, SF, ENG, FIX and SCI are similar to the register functions. They are also placed in program memory in the same format with an instruction and an argument byte.

Register, flag and display format instructions all have the following bit structure: \( bbbb \ bbbb \ iaaa \ aaaa \)

Each letter stands for one bit of the code. The b's signify the instruction byte, the i means indirect (if it is one) and the 7 a's stand for the argument.

Label Instructions

We need to distinguish between numeric and ALPHA labels. The numeric ones are coded like register instructions except that there is no "indirect". Again the labels 0 to 14 are coded as one byte instructions.

The bit structure of an ALPHA label is:

1100 \( bbb \ rrrrrrrr \) 1111 \( nnnn \ kkkkkkkk \) tttt tttt ...

In the field \( bbbrrrrrrrr \) the distance to the highest label or END in memory is stored. \( bbb \) shows the byte difference and \( rrrrrrrrr \) the register difference.

All ALPHA labels and END's are linked together into a global chain. In every global instruction (ALPHA labels and END instructions) the distance to the preceding global instruction is stored.
The operating system administers these links by keeping them up to date when you write or change any program. When the operating system searches for a program it starts searching the global chain with the **.END.** instruction, again demonstrating the significance of the **.END.** instruction.

The end of the chain is found when a global instruction with the link value of "0" is found.

As the operating system thinks that all links are correct, the careless change of any data in the link can cause a "MEMORY LOST" or lock up your calculator.

**What do these links look like:** When you calculate the difference between the addresses of the first two bytes of two neighboring global instructions and express these in bytes and registers, you get the information which is stored in `bbbr rrre rrrr array`. In our example (4 bytes and 1 register) this would be bin `1000 0000 0001` or hex `108`.

The bits `1111nnnn` follow the link array. The four ones (hex F) signify a character string (or rather, the byte hex Fn; see also text instructions!). This means that we are dealing with an ALPHA label (and not an END instruction). The `nnnn` array indicates the number of letters in the label plus one, which follow this third byte. The first letter of this chain is reserved for the possible key assignment byte or keycode. The following bytes are the ASCII codes of the letters in the name. Thus the program line `LBL "ABCD"` consists of:

```
hex C0 00 F5 00 41 42 43 44
```

(here the distance and key code arrays are set to 0.)
END Instructions

The general bit structure of an END instruction is:

1100 bbr rrrr rrrr 00e0 xpdx

As already explained in the section "Label instructions", the first 4 nybbles of this instruction are identical to those of an ALPHA label. The next eight bits though have a totally different meaning:

- **e** bit: This bit is usually a 0. Only the .END. instruction sets this bit.
- **p** bit: In a newly written program this bit is set in the END to indicate whether the program is packed. A packed program has all the null bytes removed, or "PACKed" away.
- **d** bit: This bit is used by the operating system to signify that a program has been changed.

GTO and XEQ Instructions

Here too, we must distinguish between numeric and ALPHA arguments.

ALPHA GTO's and XEQ's have a simple structure:

- **GTO**: bin 0001 1101 1111 nnnn
- **XEQ**: bin 0001 1110 1111 nnnn
The first byte indicates whether we are dealing with a GTO (hex 1D) or an XEQ (hex 1E) instruction. The second byte (hex Fn) indicates, similar to an ALPHA LBL, that a character string follows. These n letters indicate the name of the ALPHA LBL to which we want to jump. The instruction contains no information about the distance to the ALPHA LBL and no information about a possible key assignment!

If, during a running program, an ALPHA GTO or XEQ is executed, the operating system will first search the global chain starting at the .END., up to the topmost label in the chain. If the corresponding ALPHA label is not found, the operating system will search for the instruction in catalogs 2 and 3.

Now to the instructions GTO and XEQ with numeric arguments. Their structure is:

Two byte GTO: bin 1011 n'n'n'n'dbbb rrrr
Three byte GTO: bin 1101 bbbr rrrr rrrr dnnn nnnn
XEQ: bin 1110 bbbr rrrr rrrr dnnn nnnn

You will surely recognize some of these arrays. The nnnn array of the two and three byte instructions contains the label number that program execution will branch to. In the two byte GTO this corresponds to the n'n'n'n'array, only here we have the label number plus 1.

When first executing one of these instructions, the b, d and r bits are set to 0. The calculator starts to search for the corresponding numeric label. As soon as this is found, the distance to the label is stored in the b and r bits. The d bit is the direction bit, one for backwards and zero for a forwards jump. Program execution continues at the label. The next time the leap instruction is executed, the operating system will know that the jump distance is already calculated, and therefore must not search for the label again. It branches directly to the calculated place without checking to see if the employed label is the correct one, or if one even exists at all!
The jump distance is measured from the byte containing the first part of the distance code to the byte which is directly in front of the label.

The short form GTO instruction is the two byte GTO. On the one hand program memory is saved, but on the other hand there is less space for the leap distance than in the three byte GTO, namely only 4 bits. This corresponds to a maximum leap of $15 \times 7 + 6 = 111$ bytes ($bbb\ rrrr = 111\ 1111$). This holds true for labels 0 to 14.

Text Instructions

All text instructions start with a byte of the format bin 1111 nnnn. The nnnn array indicates how many letters are in the text instruction, with a maximum of 15 letters.

How are letters stored? Corresponding to the *American Standard Code for Information Interchange* one byte is used for each letter. The abbreviation for this is ASCII.

Using the CCD-Module lower case mode it is possible to enter a byte of any value (this mode will be explained in detail later).

The byte hex 7F, depending on its position in the text string, has a different meaning. If it immediately follows the text byte (hex Fn), it is not interpreted as a letter, instead indicating that the letters are to be appended to the ALPHA register without first erasing it. At any other place in the text string this byte will be interpreted as a "lazy T".
Number Input

If a number is entered as a program line, each digit has a corresponding byte in program memory. Each of these bytes represents a one byte instruction, which simulates the manual pressing of a key:

<table>
<thead>
<tr>
<th>hex code</th>
<th>digit keystrokes</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>19</td>
<td>9, decimal comma</td>
</tr>
<tr>
<td>1A</td>
<td>EEX key</td>
</tr>
<tr>
<td>1B</td>
<td>CHS key</td>
</tr>
</tbody>
</table>

Just as manual number input is closed by pressing the ENTER key or by the execution of some other function, the operating system will treat numeric input from a program line as terminated as soon as a byte is read having a value other than hex 10 to 1C.

Instructions from Plug In Modules

The possibility of extending the instruction set of the HP-41 by plug in modules represents one of the strong points of this calculating system. How are these extra instructions coded? This calls for a detailed explanation of the behavior of the operating system:
As mentioned before, the HP-41, besides having RAM (for data), also possesses a ROM address space for the operating system and plug in modules. Just like RAM, this ROM has a predefined address area, which is divided into 16 sections of identical size (4k blocks). The first 8 sections are occupied by the operating system and several hard-addressed modules: TIME, Printer and HP-IL module. The other 8 sections are assigned to the ports, each getting two sections (max. 8k byte). Therefore a module always occupies exactly one or two of these 4k blocks. If a module is plugged into the calculator, the microprocessor is able to read instructions out of the corresponding addresses. Otherwise the ROM area will seem empty.

The first two bytes within such a plug in ROM have a special meaning. They are the identity number and the number of functions within this module. The following bytes form the function list of the module, and then begins the actual function coding. Thus each function is clearly identified by two numbers:

- by the identity number of the module containing the function
- by its place in the module list: the function number.

These two numbers comprise the familiar XROM number. This XROM number appears in program memory instead of a ROM function when the corresponding module has been removed from the calculator.

Since the calculator can display these two numbers without the module being plugged in, we can be sure that they are contained in the code for the corresponding function. Furthermore we know that these numbers characterize the function sufficiently and do not expect further information in the instruction. Now for the structure:

\[ \text{bin } 1010 \ 0\text{iii } iiff \ ffff. \]

The \( i \) array indicates the ROM-ID, and the \( f \) array the function number. The ROM-ID and numbers of the functions are listed in the module handbooks. For example:
The function RNDM from the CCD-Module has the number 4 and the module ID is 9. Thus:

\[ \begin{align*}
iiii & = 01001 \\
ffff & = 000100 \\
\end{align*} \]

\textbf{XROM 09,04} = bin 1010 0010 0100 0100 = hex A2 44

This coding allows 31 different ROM-IDs (The ID = 0 is not allowed!) and 64 functions per ROM.

\textbf{Summary}

Besides needing instruction codes, some functions also need information about a register address, a numeric label or such. Using the CCD-Module it is possible to build the single instruction codes by filling in the appropriate parameters. The following paragraphs will hopefully answer a variety of questions, and others may be solved simply by trying them out! The CCD-Module contains the necessary functions to execute this in a simple way.

\textbf{Program Line Numbering}

When talking about the structure of each instruction nothing was indicated for the program line number as these are not coded in the same way. A program, whose address is fixed by the global chain, is always interpreted starting at line 01. From that point on the operating system must keep track of where an instruction starts and how many bytes it needs. The leap distances stored in GTO's and XEQ's naturally support this regulation during program execution. This means that each byte must, from the beginning, be looked at as the first or postfix byte of a multiple byte instruction. If we are moving forwards in program memory, this poses no problem. But if we are moving backwards (for example with BST), the operating system must calculate where the preceding step starts.
Instructions in Plug In Modules (Parameter)

Why do programmable instructions from plug in modules not have parameters, as, for example, the STO instruction? If the operating system recognizes an XROM instruction, it supposes that it is a two byte instruction. Of course it would have been possible to plan an extra parameter. Since not every instruction is to be extended by a parameter, it would have to be possible to distinguish between those with and those without, even if the corresponding module is not plugged in, because otherwise the extra argument byte would be interpreted as a new program line and therefore cause confusion when counting the program lines (see above). Unfortunately, a distinction like this is not planned in the operating system; moreover there would hardly be any space for this in the XROM instructions.

Key Assignments (Design and Inner Structure)

Key assignments are of two different types:

- assignments of USER programs in the main memory
- assignments of functions and programs of plug in modules and mainframe functions.

Assignments of programs which are located in the main memory store the keycode in the fourth byte of the corresponding LBL. Therefore the information is not lost when saving a program to cards, tape, or extended memory.

Assignments of functions and programs of plug in modules (XROM numbers) store the assignment information in the key assignment registers. These key assignment registers start in the main memory at address 192 (hex 0C0). As soon as new assignments are made the old ones are pushed upward (to the higher addresses), and the new assignment is now at address 192. Thus, since registers from the main memory are needed, the number of free registers which are left for programming is diminished by one register per two key assignments.
To show the layout of the key assignment registers, we will look at the following example:

First we assign the function BEEP to key -11 and the function SAS (XROM 09,05) to key 15. These assignments can later be checked with CAT'6. If we now decode the register at address 192 with the key sequence 192 PEEKR DCD ALPHA, we can see the following ALPHA characters (the bytes are represented hexadecimally):

F0 A245 41 0486 09

or to be more exact:

F0 : recognizing byte for key assignment registers (always F0)
A245 : code of the two byte function SAS (XROM 09,05)
41 : key code of key 15
0486 : code of the function BEEP. The byte 04 serves only as filler byte and is "prefix" of all one byte functions from CAT 3
09 : key code of the key -11

After erasing the key assignment BEEP on key -11, the contents of the key assignment register change as follows:

F0 A245 41 0486 00

We can see, when erasing a key assignment, that only the key code is set to 00. By this means the operating system knows that the key assignment is not active anymore. Still the plain existence of key assignment registers does not suffice for the calculator to recognize such. To be able to quickly recognize these assignments, so called "key assignment bits" are contained for all keys in the status registers * and e. When pressing a key, the operating system of the HP-41 first checks if the corresponding key assignment bit is set. If this is the case, the assigned function becomes active (only in USER mode!). If the bit is cleared, it does not even search for a possible assignment in the key assignment registers and program labels. This allows for a fast distinction of whether an assignment is present or not.
I/O Buffers (Design and Inner Structure)

I/O usually stands for input and output. Here it means 'dialog with RAM memory while avoiding the operating system'. The I/O buffer may be used by plug in modules. Some modules for example, the TIME and CCD-Module, each construct an I/O buffer and manage it independently. The I/O buffer appears to the operating system as a closed register block.

Each I/O buffer is identified by the base register which, is the lowest numbered register in the block. The four nybbles at the very right contain the most important information:

Base register: hex ii ll........

A copy of the buffer ID number is contained in nybbles 12 and 13 (ii), an ID number between hex 1 and hex E (hex F is reserved for the key assignment registers) is allowed. The two nybbles ll indicate the length of the buffer in registers.

When switching on the calculator, the operating system searches for buffers. If one is found, it erases the ID in nybble 13. Then it jumps to the register above the buffer and checks if there is another buffer there, and so on. If no more buffers are found, it branches into the plug in modules, which can reclaim their buffers by restoring nybble 13. Once all of the modules are checked we branch back into the operating system, which now erases the unclaimed buffers using a special PACK-I/O routine, and packs the buffer registers. This elucidates why, if the appropriate module is not plugged in when the calculator is turned on, the I/O buffer is erased.
Chapter 2

Operating System Enhancements
Contents Chapter 2

Operating System Enhancements

The Catalogs ................................................................. 2.05
ROM Memory Configuration ......................................... 2.10
The Functions ASN and XEQ ........................................... 2.11
  ASN ........................................................................ 2.11
  XEQ ....................................................................... 2.12
Direct and Indirect Memory Access Functions ............... 2.13
Physical Address Structure of the HP-41 Status
  Registers ................................................................. 2.14
The Input of Any Alpha Character String ....................... 2.15
Enhancements of the Operating System

In contrast to the application modules commonly used with the HP-41, the CCD-Module expands the operating system of the calculator; thus as soon as the module is inserted, in addition to the CATalog 2 functions provided by the module, several functions "native" to the HP-41 are expanded in their scope and utility. In particular, the module provides the HP-41 with additional CATalogs and catalog functions, improvements for the XEQ and ASN functions, the capability of executing and programming synthetic functions directly from the keyboard and an enhanced alpha mode for the input of lower case and special characters (these extensions for the operating system are not available in very early HP-41s; see appendix entitled Compatibility).

The Catalogs

With the CCD-Module plugged into the calculator, the three previously existing CATalogs (6 in the HP-41CX) are expanded to a total of 16, and their functions are considerably enhanced. All of the new catalogs may be halted during execution by R/S, and subsequently stepped through in either direction using SST or BST. In contrast to the operation of the native catalogs of the HP-41, the SHIFT annunciator remains lit during the use of BST. The key sequence SHIFT R/S will even cause the catalog listings to be run in reverse. A running catalog may be speeded up by pressing any key other than R/S or ON. The catalogs will now be individually described.

CAT'0

CAT'0 shows the ID or AID of all devices in the Hewlett-Packard Interface Loop if any are present. When the catalog is stopped the displayed device can be selected by pressing ENTER, if you press C, and the selected device is displayed, a Selected Device Clear message will be sent to that device.
Pressing the back arrow key terminates the stopped catalog. When there is no HP-IL module in the calculator, the message "NO HPIL" is displayed.

**CAT'1**

This catalog executes the normal CAT 1 of the HP-41 with no enhancements in the manner of execution. More information can be found in the HP-41 handbook.

**CAT'2**

This catalog is greatly enhanced in its operation in comparison to the standard CAT 2 of the HP-41. When it is first executed only the "headers" of each ROM are displayed (like the HP-41CX). If the catalog is halted with R/S the user may press ENTER to view the function block of the currently displayed "header". When the desired function is located, it may be executed directly from the catalog by pressing XEQ (the function will be inserted in program mode), or the function may be assigned to a key by pushing the A key. A second press of the ENTER button returns you to the catalog listing of only the ROM "headers".

**CAT'3**

This catalog executes the normal CAT 3 of the HP-41. There are no changes in the manner of execution. More information can be found in the HP-41 handbook.
CAT'4

Like the function EMDIR of the Extended Functions module and CAT 4 of the HP-41CX, CAT'4 displays the names lengths, and types of all files in extended memory. It has the additional feature of displaying the three additional file types used by the CCD-Module. The three file types are, I/O buffers (displayed as B), matrices (M), and key assignment files (K). If no extended memory is present the error message "NO XF/M" is displayed.

CAT'5

Executes the function ALMCAT of the TIME module. When there is no TIME module in the calculator the message "NO TIMER" is displayed.

A note regarding use of this instruction: If you have an older HP-41 equipped with an early revision of the 82104A card reader, and if no alarms exist, the calculator will appear to lock up with "CAT'5" in the display. This is not a system failure but a trivial quirk of the older machine; the next keystroke will cause the message to disappear and the pressed key will be executed.

CAT'6

CAT'6 shows all key assignments in keycode order, starting at the sigma + key and working its way horizontally and then dropping down to the next row etc.. On the right side you will see the keycode and on the left side the function will be displayed. Even synthetic key assignments ("RCL M", "TEXT 7") are shown correctly and not as an XROM number. Pressing "C" deletes the shown key assignment when the catalog is stopped. When there are no key assignments the message "NO KEYS" is generated.
CAT'7

Executes the function **DIR** of the HP-IL module. For a detailed description of this function see the owners handbook for the HP-IL module. When there is no HP-IL module in the calculator, the message "NO HPIL" is shown.

CAT'8 - CAT'F

These catalogs operate in a manner similar to the enhanced CAT'2 function of the CCD-Module, except that each of these catalogs addresses a single "ROM page" of the I/O ports of the HP-41. Both the catalogs and the ROM pages are numbered from 8 to F; as one might therefore expect, each of the catalogs in this group has a number identical to the ROM page whose contents it examines. The I/O ports are addressed as follows:

- Port 1: Page 8 and 9
- Port 2: Page A and B
- Port 3: Page C and D
- Port 4: Page E and F

Each port of the HP-41 can thus be occupied by up to 8 Kbytes of program material. Since most application modules address the lower 4K of the port in which they reside (if it is a 4K application module), then the upper page of that I/O port is inaccessible under normal circumstances, and the appropriate catalog will display the message "NO ROM" for that address block.

The operating system of the HP-41 addresses 16 4k Byte pages. They are used in the following manner:

<table>
<thead>
<tr>
<th>Page</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>operating system (System ROM 0)</strong></td>
</tr>
<tr>
<td>1</td>
<td><strong>operating system (System ROM 1)</strong></td>
</tr>
<tr>
<td>2</td>
<td><strong>operating system (System ROM 2)</strong></td>
</tr>
<tr>
<td>3</td>
<td><strong>Not used by HP-41C and CV. Extended operating system of the HP-41 CX (System ROM 3)</strong></td>
</tr>
<tr>
<td>4</td>
<td><strong>Service Module or disabled IL printer</strong></td>
</tr>
</tbody>
</table>

(C) Copyright W&W Software Products GmbH 1985
5 Used for the TIME module in the HP-41 C and CV. Extended operating system in the CX (system 5a and 5b with bank switching)

6 Used for by the printer ROM

7 Used for the HP-IL module (note: the printer ROM is contained in the HP-IL module, but can, using a certain switch, be put on the address area 4, and therefore be switched off)

8 Port 1 lower 4 kByte
9 Port 1 upper 4 kByte
A Port 2 lower 4 kByte
B Port 2 upper 4 kByte
C Port 3 lower 4 kByte
D Port 3 upper 4 kByte
E Port 4 lower 4 kByte
F Port 4 upper 4 kByte

The read/write memory of the main and extended RAM expansion modules is managed in a different manner from ROM, and is not addressed to the port in which it occupies; thus it is possible to have all of your memory modules built into the HP-41, leaving the four I/O ports free for application packs. One could even have the CCD-Module installed internally and addressed to port 3, leaving that port free for the HP-IL or printer modules; since the addresses of these ROMs are fixed internally by the HP-41, other modules may be inserted in the "ports" they occupy if special electronic modifications are arranged to enable this (for more information on this subject contact any W&W Software Products division).
ROM Memory Configuration

Page

<table>
<thead>
<tr>
<th>F</th>
<th>E</th>
<th>D</th>
<th>C</th>
<th>B</th>
<th>A</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORT 4</td>
<td>PORT 3</td>
<td>PORT 2</td>
<td>PORT 1</td>
<td>HP—IL Module <em>(MASS ST)</em></td>
<td>Printer Module</td>
<td>Time Module</td>
<td>CX : —EXT FCN</td>
<td>Reserved</td>
<td>CX : —EXT FCN</td>
<td>Operating system</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Switched bank
The Functions ASN and XEQ

To understand the following you should carefully read the section entitled *Functions for advanced programming* or have at least basic knowledge about synthetic programming.

**ASN**

The enhanced ASN function permits the following keyboard entries:

a) **The normal ASN function:** If the user presses ASN followed by ALPHA, the standard ASN function of the unenhanced HP-41 will be run.

b) **The assignment of any two byte function:** When you press the ASN key with the CCD-Module present you will see the following prompt: "ASN: ___: ___". The calculator is prompting for two decimal byte values. When you key in two bytes and press any key after that the two byte function is assigned to that key. If you first press the H key the operating system will prompt for hexadecimal values. With ENTER or "." you return to the decimal prompt.

c) **Assigning an XROM number:** The ASN function of the CCD-Module allows the assignment of XROM numbers without the module with that XROM number in the calculator. After you have pressed ASN you simply press XEQ and you will see: "ASN XROM: __". This prompt initially requests input of the ROM ID (i.e., the portion of the XROM number which precedes the comma, such as 9 or 11 for the CCD-Module). After the entry of these digits, the prompt becomes "ASN XROM:09: ___", which indicates that input of the function ID, which is to say, the portion of the XROM number which follows the comma, is now expected (e.g. 01, for the function "B?" of the CCD-Module). The prompt becomes "XROM:09:01 __", which requests input of the code for the key to which the function is to be assigned.
The enhanced XEQ function allows the following keyboard entries:

a) **The normal XEQ function:** If ALPHA or a number key is pressed after XEQ, we obtain the normal XEQ function. It is the same as the unenhanced XEQ function of the HP-41.

b) **The execution of any two byte function:** When you press XEQ and then ENTER you will see the prompt "XEQ:__:__". The calculator is prompting for two decimal byte values. When you key in two values the function is executed or inserted into a program. If you press H before keying in any value you will see "XEQ:"__"__" and the calculator prompts for a hexadecimal input. By pressing ENTER or "." you are returned to the decimal prompt.

c) **The execution of an XROM number:** The expanded XEQ function provided by the CCD-Module also permits the execution of function or application programs by XROM number, even if the module is not present in the HP-41. The key sequence XEQ ENTER XEQ generates the prompt "XEQ XROM:__"; this initially the input of the ROM ID number (the portion of the which precedes the comma, such as 09 or 11 for the CCD-Module). After the entry of these digit, the prompt becomes "XEQ XROM:09:__", which indicates that the input of the function ID, which is to say, the portion of the XROM number following the comma, is now expected (e.g., 01, for the function B? of the CCD-Module). If the calculator is in run mode and the appropriate module is present (the CCD-Module in our example), the function is immediately executed; otherwise the error message "NONEXISTENT" is displayed. If the HP-41 is in program mode then the instruction is inserted as a program line.

**NOTE**

To avoid confusion, throughout this manual, and in the programming of the CCD-Module, the appearance of the colon (:) preceding and input prompt indicates that the number to be input is of the decimal form, and if the colon is replaced by an apostrophe ('), the input is in the hexadecimal format.
Direct and Indirect Memory Access Functions

To simplify the insertion of synthetic program lines, the CCD-Module provides the capability for the direct entry of synthetic instructions. Note that for very early HP-41's these capabilities are not available, so one must use synthetic key assignments if this capability is desired from a very early machine (see appendix on compatibility).

All memory access functions (RCL, STO ,X<> ) can now be accessed directly from the keyboard and used to address all of the status registers of the HP-41. Thus access to and manipulation of the contents of registers M, N, O, P, Q, a, b, c, d, e, and lazy T is now no more complex than working with X, Y, Z, T, and L. The keystroke sequence used to apply these functions to the status registers is RCL . d.

Exercise caution in manipulating status register contents:

Altering the contents of registers L and a through e can lead to a "MEMORY LOST" condition or to a system crash if the register contents are improperly altered. Alteration of the "cold start constant" 169 in register c will always result in "MEMORY LOST". Before experimenting with these registers the user should throughly familiarize himself with the theory and practical applications of synthetic programming. Several of the references listed in the bibliography provide excellent discussion on the subject.
Physical Address Structure of the HP-41 Status Registers

<table>
<thead>
<tr>
<th>Byte:</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Shifted key assignment flag bits**
- **Scratch**
- **Line number**
- **Flag register**
- **Program return stack**
- **Unshifted key assignment flag bits**
- **Scratch**
- **ALPHA - Register**
- **Stack register L**
- **Stack register X**
- **Stack register Y**
- **Stack register Z**
- **Stack register T**

Status registers

(C) Copyright W&W Software Products GmbH 1985 2.14
The Input of Any Alpha Character String

The CCD-Module enables the user to place in the ALPHA register, or enter directly into a program line, any of the 256 character bytes available on the HP-41 (see byte table, pages 1.20 - 1.21). This was previously possible using the Extended Functions module or synthetic programming techniques. Previously undreamed of possibilities, exploiting the use of direct entry of lower case and special characters, are presented especially to programmers making extensive use of printers and HP-IL peripherals. The short sample programs which follow make it apparent as to the extent to which memory space may be conserved by taking advantage of these functional enhancements:

Printout of the Text Line "Hewlett-Packard"

Using standard programming techniques

```
01 LBL "HP"
02 "H"
03 ACA
04 SF 13
05 "EWLETT-
```

Using the lower case mode of the CCD-Module

```
01 LBL "HP"
02 "Hewlett
03 AVIEW
04 END
```

```
06 ACA
07 CF 13
08 "P"
09 ACA
10 SF 13
11 "ACKARD"
12 ACA
13 CF 13
14 PRBUF
15 END
```

```
PLNG "HP" 26 BYTES
```

```
PLNG "HP" 46 BYTES
```

(C) Copyright W&W Software Products GmbH 1985
Sending an ESC-Sequence to the Think Jet Printer

using standard programming techniques by exploiting the lower case mode of the CCD-Module

01·LBL "WIDE"
02 27
03 ACCHR
04 38
05 ACCHR
06 107
07 ACCHR
08 49
09 ACCHR
10 83
11 ACCHR
12 END

PLNG "WIDE"
32 BYTES

Thus it is readily apparent that the direct entry of lower case and special characters greatly facilitates the ease and "byte economy" in programming.

The lower case and special character entry mode of the CCD-Module is available in ALPHA mode when user mode is off. The special keyboard overlay included with your CCD-Module has the printer control characters and other special characters listed according to the color code which is described in the table on the next page.
User Mode on  Normal Alpha keyboard is active
Blue letters on key faces Capital letters A-Z and some special characters
Blue letters overlay Special characters available by pressing SHIFT
User mode off Lower case mode active
Red letters on overlay Special characters for unshifted keys (note: that there are no special notations for the lower case letters, which have normal key locations).
Green letters on overlay Special characters available by pressing SHIFT

If the desired character is not available on the keyboard you can key it in by its decimal or hexadecimal value by pressing SHIFT ENTER for the decimal, or press SHIFT ENTER H if you want to enter the character in hex.

Warning:
It is not possible to cancel this byte prompt. If you see the prompt, just key in any character (except zero) and delete it afterwards. If the lower case mode is not desired it can be suspended by using the program TLC (Advanced Programmers Functions).

When a special character is also a printer or IL device control code it is shown on the right side of the key with its control code function name (see the overlay included with the CCD-Module).

Please note:
In version -W&W CCD A you must not key in a space in a program line while your are in lower case mode because program memory could be altered. If you want to key a space in a program line either turn the lower case mode off or use the byte prompt!
Chapter 3

Functions from Catalog 2
Contents Chapter 3

Functions Appearing in Catalog 2

B? ................................................. 3.05
CLB ............................................... 3.07
SAS ............................................... 3.08
CAS ............................................... 3.09
RNDM ........................................... 3.10
SEED ............................................ 3.12
SORT ............................................ 3.13
Functions from Catalog 2

B?

B? is used when you wish to determine whether a given I/O buffer exists. The buffer ID number, a decimal number from 1 to 14 which identifies the ROM which generated the buffer, is expected in X. If the function is executed as part of a running program, the program line immediately following it is executed if the buffer is present; the step is skipped if the buffer does not exist. If B? is executed from the keyboard, then a YES or NO is displayed depending on whether the buffer is present in the machine. This behavior is analogous to standard HP-41 conditional functions such as, X=Y? . If the absolute value of the integer portion of X is greater than 14 then a "DATA ERROR" message is generated.

Many modules create I/O buffers for intermediate data storage. For these I/O buffers the free memory space between the key assignments and the last program (starting with the .END. ) is used. The room used by the I/O buffers is taken away from space available for programs. The following I/O buffers are known up to now:

<table>
<thead>
<tr>
<th>Module</th>
<th>ID</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAVID-ASSEM</td>
<td>1 and 2</td>
<td>Where the program counter is and to save labels. Buffer 2 is for jumps to nonexistent labels.</td>
</tr>
<tr>
<td>CCD-Module</td>
<td>5</td>
<td>Wordsize, random number, active matrix</td>
</tr>
<tr>
<td>HP Advantage ROM</td>
<td>5</td>
<td>Same as the CCD-Module Matrix functions</td>
</tr>
<tr>
<td>TIME Module</td>
<td>10</td>
<td>Time alarms</td>
</tr>
<tr>
<td>Plotter Module</td>
<td>11</td>
<td>Intermediate storage for plotter and bar code instructions</td>
</tr>
<tr>
<td>HP-IL Development</td>
<td>12</td>
<td>Scope Mode and used for saving HP-IL commands</td>
</tr>
<tr>
<td>CMT 300</td>
<td>13</td>
<td>Used for I/O of the CMT measurement system</td>
</tr>
<tr>
<td>DATAFILE ROM</td>
<td>14</td>
<td>Active data file</td>
</tr>
</tbody>
</table>

(C) Copyright W&W Software Products GmbH 1985
Input

X register: \( a_a \) (Buffer ID)

Example

We wish to determine if the Time Module has an active alarm buffer. This is accomplished by using the keystroke sequence

10 B?

If the buffer exists you will see a "YES" in the display, otherwise the message "NO" will be displayed.

Further Hints

The function B? uses only the absolute integer part of the number in the X register. Executing \(-5.678 \text{ B?}\) will show the same result as \(5 \text{ B?}\)

Related Functions

CLB, GETB, SAVEB
CLB

The function **CLB** clears an I/O buffer. The buffer ID number (between 1 and 14) is expected in X.

Input

X register:  *aa* (Buffer ID)

Example

In order to create more free registers in RAM we wish to clear the time alarms currently in our HP-41. We do this with the key sequence

10 CLB

Further Hints

The function **CLB** uses only the absolute integer part of the number in the X register. Executing -5.678 CLB will show the same result as 5 CLB. If the specified buffer does not exist no error message is generated.

Related Functions

**B?, GETB, SAVEB**
SAS

This sets the autostart flag of the CCD-Module. When this flag is set, the HP-41 begins program execution from its current position in program memory whenever the calculator is turned on using the ON key. Its operation is different from user flag 11 which is cleared every time the HP-41 is turned on and must be set each time before the calculator is turned off. With SAS one can turn off the calculator without interrupting a program, since execution will resume at the next step when the HP-41 is turned on.

Input

none

Example

The following program must not be interrupted even by toggling the calculator off then on with the ON key. You only have to put the SAS function in your program and if you turn the calculator off and on again your program will continue from the same location. In our example the program may not be stopped when PMTK is executed.

01 SAS
02 "YES/NO YN"
03 PMTK
04 CAS
05 END

Further Hints

The flag that controls this function resides in byte 4 of the c status register. It uses bit 6 of this byte. Therefore this flag is not one of the 56 standard HP-41 user flags.

Related Function

CAS
CAS

The function **CAS** (clear autostart) clears the autostart flag of the CCD-Module.

**Input**

none

**Further Hints**

The CCD-Module autostart flag is not in flag register d but is in status register c (Reg. 13, Byte 4, Bit 6).

**Related Function**

SAS
RNDM

The function **RNDM** creates a random number which is written into the X register. The random number has a value between 0 and 1.

**Input**

none, but you can create a starting value with the function **SEED**.

**Examples**

With a starting value of 0.1 the following random numbers are generated:

```
0.1 SEED
RNDM 0.311327
RNDM 0.753794
RNDM 0.222201
RNDM 0.447348 etc.
```

**Further Hints**

The most recent random number will be saved in the I/O buffer of the CCD-Module. Any new random numbers will be created from the previous one using the following Algorithm: \( X = \text{FRC} (\text{Buffer} \times 9821 + 0.211327) \). If there is not enough memory for the I/O buffer, the message "NO ROOM " is displayed.
Caution:

In the version -W&W CCD A of the CCD-Module values output by RNDM which are less than 0.1 are not normalized correctly. When you use the random number instantly there will be no mistake, but if it is to be saved in a data register you must use FRC right after RNDM because otherwise all random numbers smaller than 0.1 are changed to zero.

Related Function

SEED
SEED furnishes an initial value, which is stored in the I/O buffer of the CCD-Module, for the computation of a random number using the function RNDM. Only the fractional part of the X register is used for the starting value.

Input

X register: starting value, only the part to the right of the decimal point is used.

Further Hints

With CLX SEED you can clear the random number register from the buffer, thus freeing one more register for program storage.

Related Function

RNDM
SORT

This function sorts the contents of registers starting with register $R_{iii}$ up to register $R_{jjj}$. Alpha Data may also be intermingled with the data to be sorted. The largest value will be stored in register $R_{jjj}$. By choosing clever parameters you can either sort high to low or low to high.

Input

X register: $iii.jjj$

Examples:

The contents of the registers 1-10 will be sorted first low to high and then high to low. The original order of the register contents is

\begin{align*}
R01 &= 0.1356 \\
R02 &= 3.5462 \\
R03 &= 9.7363 \\
R04 &= 2.4138 \\
R05 &= 7.8467 \\
R06 &= 4.0629 \\
R07 &= 4.0506 \\
R08 &= 2.9577 \\
R09 &= 9.2921 \\
R10 &= 0.1220
\end{align*}
1) 1.010 SORT  

<table>
<thead>
<tr>
<th>R01=</th>
<th>0.1220</th>
<th>R01=</th>
<th>9.7363</th>
</tr>
</thead>
<tbody>
<tr>
<td>R02=</td>
<td>0.1356</td>
<td>R02=</td>
<td>9.2921</td>
</tr>
<tr>
<td>R03=</td>
<td>2.4138</td>
<td>R03=</td>
<td>7.8467</td>
</tr>
<tr>
<td>R04=</td>
<td>2.9577</td>
<td>R04=</td>
<td>4.0629</td>
</tr>
<tr>
<td>R05=</td>
<td>3.5462</td>
<td>R05=</td>
<td>4.0506</td>
</tr>
<tr>
<td>R06=</td>
<td>4.0506</td>
<td>R06=</td>
<td>3.5462</td>
</tr>
<tr>
<td>R07=</td>
<td>4.0629</td>
<td>R07=</td>
<td>2.9577</td>
</tr>
<tr>
<td>R08=</td>
<td>7.8467</td>
<td>R08=</td>
<td>2.4138</td>
</tr>
<tr>
<td>R09=</td>
<td>9.2921</td>
<td>R09=</td>
<td>0.1356</td>
</tr>
<tr>
<td>R10=</td>
<td>9.7363</td>
<td>R10=</td>
<td>0.1220</td>
</tr>
</tbody>
</table>

2) 10.001 SORT

Further Hints

When ALPHA and numerical data are sorted together, ALPHA data is considered greater than numerical data.

Related Function

SORTFL
Chapter 4

Matrix Functions
## Contents Chapter 4

### Matrix Functions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>4.05</td>
</tr>
<tr>
<td>Organization and Construction of Arrays</td>
<td>4.06</td>
</tr>
<tr>
<td>Functions for the Construction of Data Arrays</td>
<td>4.09</td>
</tr>
<tr>
<td>MDIM</td>
<td>4.09</td>
</tr>
<tr>
<td>DIM</td>
<td>4.13</td>
</tr>
<tr>
<td>Functions for the Manipulation of Element Pointers</td>
<td>4.14</td>
</tr>
<tr>
<td>?IJ</td>
<td>4.14</td>
</tr>
<tr>
<td>?IJ=A</td>
<td>4.15</td>
</tr>
<tr>
<td>IJ=</td>
<td>4.16</td>
</tr>
<tr>
<td>IJ=A</td>
<td>4.17</td>
</tr>
<tr>
<td>Input and Output Functions for Data Arrays</td>
<td>4.18</td>
</tr>
<tr>
<td>&gt;C+</td>
<td>4.19</td>
</tr>
<tr>
<td>&gt;R+</td>
<td>4.21</td>
</tr>
<tr>
<td>C&gt;+</td>
<td>4.23</td>
</tr>
<tr>
<td>C&gt;–</td>
<td>4.25</td>
</tr>
<tr>
<td>R&gt;+,</td>
<td>4.27</td>
</tr>
<tr>
<td>R&gt;–</td>
<td>4.29</td>
</tr>
<tr>
<td>Functions for Shifting and Exchanging Elements</td>
<td>4.31</td>
</tr>
<tr>
<td>C&lt;&gt;C</td>
<td>4.31</td>
</tr>
<tr>
<td>R&lt;&gt;R</td>
<td>4.33</td>
</tr>
<tr>
<td>MOVE</td>
<td>4.35</td>
</tr>
<tr>
<td>SWAP</td>
<td>4.37</td>
</tr>
<tr>
<td>Functions for Determining the Extreme Values of Array Elements</td>
<td>4.39</td>
</tr>
<tr>
<td>MAX</td>
<td>4.39</td>
</tr>
<tr>
<td>MAXAB</td>
<td>4.41</td>
</tr>
<tr>
<td>CMAXAB</td>
<td>4.43</td>
</tr>
<tr>
<td>RMAXAB</td>
<td>4.44</td>
</tr>
<tr>
<td>MIN</td>
<td>4.45</td>
</tr>
<tr>
<td>PIV</td>
<td>4.46</td>
</tr>
<tr>
<td>R&gt;R?</td>
<td>4.47</td>
</tr>
<tr>
<td>Sums and Norms</td>
<td>4.49</td>
</tr>
<tr>
<td>SUM</td>
<td>4.49</td>
</tr>
<tr>
<td>SUMAB</td>
<td>4.51</td>
</tr>
<tr>
<td>CSUM</td>
<td>4.52</td>
</tr>
<tr>
<td>RSUM</td>
<td>4.53</td>
</tr>
<tr>
<td>CNRM</td>
<td>4.54</td>
</tr>
<tr>
<td>RNRM</td>
<td>4.55</td>
</tr>
<tr>
<td>FNRM</td>
<td>4.56</td>
</tr>
</tbody>
</table>
### Functions for Mathematical Manipulation of Arrays

- **M+** .......................... 4.57
- **M-** .......................... 4.57
- **M*** .......................... 4.59
- **M/** .......................... 4.61
- **M*M** .......................... 4.63
- **YC+C** ......................... 4.65
- **R-QR** ......................... 4.67
- **R-PR** ......................... 4.69

### Program Examples

- Program "ABIN" (Solution of a System of Linear Equations) .................. 4.77
- Program "INV" (Computation of an Inverse Matrix) .................. 4.83
Functions for the Construction and Manipulation of Arrays

In this section we shall describe the use of the CCD-Module in the construction and manipulation of one and two-dimensional arrays. We shall begin with two observations:

The arrays may contain numeric as well as alphanumeric data. Arrays with only numeric elements are called two-dimensional matrices. All mathematic functions can be performed only on arrays such as this. In this chapter all arrays are illustrated in square cells with borders to convey the relationships between these elements and the memory locations used in storing them. For example,

\[
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} & a_{14} \\
  a_{21} & a_{22} & a_{23} & a_{24} \\
  a_{31} & a_{32} & a_{33} & a_{34}
\end{bmatrix}
\]

Arrays can be constructed and stored in either main or extended memory. In the first case they are called RAM Arrays, and in the second they are called XM Arrays. Except some few differences in names and arrangements both kinds of arrays are the same. In Catalog 4 of the CCD-Module the XM Arrays can be recognized by the letter M (for Matrix).
Organization and Construction of Arrays

Each array consists of status information and its element values. The status information contains the array name, the array dimensions, and a pointer. A status register is set aside for RAM Arrays and it contains this information except for the name. The name is given by the position of the register: the RAM Array "R012" has its status register in register 12. Its elements are saved in the registers immediately following. So that the status information cannot be destroyed when handling register 12, (for example ST+ 12 or RCL 12) it is arranged in the form of ALPHA data. The status information of XM Arrays is contained within the file status register of the extended memory file (usually called the "file header") and therefore no extra memory is needed.

Each element is marked through its position in the array, i.e., by specifying the row and column in which it is found. We shall call these positional numbers the row and column indices, and designate them by the letters i and j. The smallest possible value for i and j is 1, the greatest possible for i is m and for j it is n. Therefore:

Row index \(1 \leq i \leq m\)

Column index \(1 \leq j \leq n\)

For all array functions: If the value 0 should be given for i or j, it is changed to 1.

To construct an array the command MDIM is used. By giving the array name (in the ALPHA register) and the array dimensions (X value= mmm.nnn) the suitable file is constructed. The dimension may be input later, however, using the function DIM.

(C) Copyright W&W Software Products GmbH 1985 4.06
The position and size of the arrays (matrices, vectors, etc.) are managed automatically using new functions provided in the CCD-Module. Therefore the user does not have to keep track of the position of single registers or rather, single elements. For example, matrix arrays can be arranged in the extended memory, and these arrays can be combined with matrix arrays that are positioned in main memory. The CCD-Module makes possible a wide variety of unified input instructions for performing these combinations.

Input Commands for Array Functions:

All CCD-Module matrix combination functions expect their input parameters in the ALPHA register. If, for instance, there are three matrices of the same size, "A", "B", and "C", which have previously been dimensioned with M DIM, then the functions for combining matrices have to obey the following rules with regard to the alpha register:

1) We want to combine matrix "A" with matrix "B" and put the resultant array in matrix "C". The input into the ALPHA register would be "A,B,C," with the operands being separated by commas. Now the desired function (for example M*M) may be executed.

2) Matrix "A" is to be combined with matrix "B" the resultant matrix is placed in array "B". The ALPHA register should look as follows: "A,B,B" or simply "A,B". If the resultant array is not mentioned, the result is placed in the second array. (Caution: When using the function M*M the resultant array should not have the same dimensions as one of the operands, as this will lead to an error message!)

3) We want to matrix "A" with itself and put the resulting array in matrix "C". The ALPHA register would look as follows: "A,A,C" or simply "A,,C". Since the second operand is identical to the first one, it does not have to be mentioned. But in no case should you forget the second separating comma. (Rule: There should always be two commas to the left of the resultant matrix unless the result matrix is the same as the second operand.)
4) We want to multiply matrix "A" with itself and put the resultant matrix into "A". So the ALPHA register contains: "A,A,A" or simply "A". This is a logical presumption after having read all of the above. If a combination function finds only one name in the ALPHA register, the operation will only be done on this array and the result put there as well.

5) If one operand is given the name "X", the other operand will be combined with the contents of the X register.

Matrices that are located in the main memory always have a name consisting of four characters. The first letter must always be "R" (for register). The next three characters consist of numbers which specify the relative address of the first register of the matrix (see the next section MDIM). In the rest of the chapter the abbreviations OPI, OP2, and RES are used as names for the data arrays (Operand 1, operand 2, result).
Functions for the Construction of Data Arrays

MDIM

MDIM is used to create one or two dimensional arrays, and to redimension them.

Input

ALPHA register: name of array

The array name is also the name of the file. For files in main memory, this name must take on the form "Rxxx", where xxx is a three digit decimal number giving the relative address of the status register, which is also the first register of the stored array (subsequent registers contain the matrix elements). For arrays in extended memory the first seven characters in alpha are used. The name "X" may not be used, since for several matrix functions it names the X register as an operand. If ALPHA is cleared when MDIM is executed, then no array has been specified, and the function operates on the current extended memory file which must be of file type M.

X register : $mmm.nnn$ (Array dimensions)

$mmm$ is the number of rows and $nnn$ is the number of columns. If either $m$ or $n$ have the value 0, it will be changed to 1.
Examples

To construct an array named "MATRIX" with 3 rows and 4 columns, in extended memory, the following key sequence is used:

"MATRIX"
3.004
MDIM

To build the same array in the main memory, starting with register 10, we execute:

"R010"
3.004
MDIM

In this case registers R11 through R22 will contain the twelve elements of the 3 * 4 array. The field created by either of these operations will have elements which we designate $a_{ij}$ to $a_{34}$, as shown below.

```
     Column
     +--------+--------+--------+--------+
     | a_{11} | a_{12} | a_{13} | a_{14} |
     +--------+--------+--------+--------+
     | a_{21} | a_{22} | a_{23} | a_{24} |
     +--------+--------+--------+--------+
     | a_{31} | a_{32} | a_{33} | a_{34} |
     +--------+--------+--------+--------+
```
Using
"MATRIX"
4.004
MDIM
the array "MATRIX" is redimensioned so it now has the elements $a_{11} - a_{44}$.

Further Hints

When a new array is created, all of its elements are set to 0. This is true for arrays both in main memory and XM. However, when an array is redimensioned, only the newly added elements are cleared; the values in the old elements constituting the original array are retained. When the dimension of an array is reduced, the values which were in the elements are now superfluous, and are lost.

The values of the array elements are saved row by row, i.e., the value of the last element of the first row is followed by the value of the first element of the second row. When redimensioning an array the sequence of the rows is preserved, although as a rule, their relative positions are altered after the redimensioning).

Example: Array "A" is formed with dimensions of 2 * 3 and contains the values 1-6 in the following order:

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a_{11}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
After redimensioning using the key sequence

"A"
3.002
MDIM

we obtain the following array

<table>
<thead>
<tr>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>a11</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>a21</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>a31</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

If an array is enlarged while redimensioning, the new elements are initialized with the value 0. If an array in XM is made smaller by redimensioning the extra elements are lost after the operation. In a RAM array only the dimension which is stored in the status register is changed, the registers and their contents are preserved.

Related Function

DIM
**DIM**

DIM is used to recall the dimensions of an array.

**Input**

ALPHA register: Array name

**Output**

X register : *mmm.nnn* (this number represents the array dimensions. *mmm* accounts for the number of rows, and *nnn* for the number of columns.)

**Further Hints**

If, while dimensioning an array, *m* or *n* have the value 0, the value 1 is used. This is shown in the output of DIM, by the following example:

"R000"
CLX
MDIM
DIM

the result is the number 1.001 in the X register.

**Related Function**

MDIM
Functions for the Manipulation of Element Pointers

?IJ

The function ?IJ places in X the current pointer position iii.jjj of the current data array (the stack is lifted).

Input

None

Example

In an existing matrix named "MAT" we shall set the element pointer to a₂, the second element of the first row, using the function IJ=A, and the keystroke sequence:

"MAT"
1.002
IJ=A

At a later point in the program we want to recall the pointer position. We do this using:

?IJ

In this case, the function ?IJ puts the value 1.002 into the X register.

Further Hints

The function ?IJ does not need an ALPHA input. The output is always the pointer position of the current data array!

Related Functions

?IJA, IJ=, IJ=A
?IJA

The function ?IJA places the pointer position \( iii.jjj \) of the specified data array into the X register (the stack is lifted).

Input

ALPHA register: Name of the data array

Example

After dimensioning a matrix we wish to determine the location of the element pointer and to make it the current array. This is accomplished by performing "MAT"
3.003
MDIM
?IJA

Now the matrix "MAT" is the current matrix. Since the value 1.001 has been placed in X, the pointer is set to \( a_{11} \) after the array was redimensioned.

Further Hints

After executing the function ?IJA the specified data array has become the current data array. The information about the current data array is stored in the I/O buffer of the CCD-Module, i.e. if there are no existing registers for this buffer, the error messages "PACKING" and "TRY AGAIN" show up. If there is no name mentioned, the error message "NAME ERR" will be displayed.

Related Functions

?IJ, IJ=, IJ=A
IJ=

Through the function IJ= the pointer of the current data array is put on the element with the row index \( iii \) and the column index \( jjj \).

Input

X register : \( iii.jjj \)

Example

The pointer of the current data array should be positioned to the element \( a_{i,j} \), so that the value of this element can be recalled. This is accomplished by

1.002
IJ=
C>+

Further Hints

If there is no existing current data array (for example because of loss of the I/O buffers) the error message "NONEXISTENT" will be generated.

Related Functions

?IJ, ?IJA, IJ=A
**IJ=A**

With the function **IJ=A** the pointer of the specified data array will be positioned to the element with the row index *iii* and the column index *jjj*.

**Input**

- **X** register : *iii.jjj*
- **ALPHA** register : Name of the data array

**Example**

After dimensioning a matrix with the name of "R010", the pointer may be set to the last element of this matrix. This happens with the following steps:

"R010"
5.005
MDIM
IJ=A

**Further Hints**

After executing the function **IJ=A** the specified data array is the current data array. Information about the current data array is saved in the I/O buffer of the CCD-Module, i.e. if there are no free registers present to create this buffer, the error messages "PACKING" and "TRY AGAIN" are generated. If no name is mentioned, the error message "NAME ERR" will be displayed.

**Related Functions**

?IJ, IJA, IJ=
Input and Output Functions for Data Arrays

>C+, >R+, C>+, C>-, R>+, R>-

These six functions are the input and output functions that the CCD-Module uses for all data arrays. Storage or recalling is always done from the current pointer position (see IJ= and IJ=A). The following nomenclature has been established for this group of functions.

A preceding ">" sign means input function: The element at the current pointer position will be filled the value in the X register.

A following ">" sign denotes an output function: The stack is lifted and the element at the current pointer position will be written to the X register. The "+" sign means that the pointer will be incremented by one position, and the "-" sign means that the pointer will be decremented one position.

The letter "C" (column) means that storing and recalling will be done columnwise, the letter "R" (row) signifies that this will be done rowwise.
The function >C+ enables us to fill in data array elements column by column, i.e. in an array with $i$ rows and $j$ columns, the value of $i$ is incremented by 1. The order of element input is shown clearly in the diagram below:

![Diagram showing order of element input](image)

**Input**

X register: value to be placed in array element (may be Alpha data!)

**Example**

The following program shows how to input the elements of a matrix columnwise.

```
01*LBL "CIN"
02 "R010"
03 CLX
04 IJ=A
05*LBL 01
06 STOP input your data here and then hit R/S
07 >C+
08 GTO 01
09 END
```
A similar program for the output of elements of a matrix is explained in the section for the function \( C>+. \)

**Further Hints**

When the last element of a matrix has been input, the pointer is positioned on a nonexistent element. Attempts to store or recall further elements will result in the error message "END OF FL".

**Related Functions**

\( >R+, C>+, C>-, R>+, R>-, \)
The function $\texttt{\textgreater R+}$ makes it possible to store data array elements row by row, thus in an array with $i$ rows and $j$ columns the value of $j$ will be incremented by one after the input of each element. The order of the element input can be clearly seen in the diagram below:

Input

X register: value to be placed in array element (may be ALPHA data)

Example

The following program shows how to input the elements of a matrix row by row:

01*LBL "RIN"
02 "R010"
03 CLX
04 IJ=A

05*LBL 01
06 STOP enter your data here and then hit R/S
07 $\texttt{\textgreater R+}$
08 GTO 01
09 END
A corresponding program for the output of elements of a matrix row by row is explained under \texttt{R>+}.

Further Hints

When the last element of a matrix has been input, the pointer is positioned to a nonexistent element. Attempts to put in or recall further elements will generate the error message "END OF FL".

Related Functions

\texttt{>C+, C>+}, \texttt{C>-}, \texttt{R>+}, \texttt{R>-}
The function \( C>+ \) allows the recalling data array elements column by column, i.e. in an array with \( i \) rows and \( j \) columns the value of \( i \) will be incremented by one after each element is output. The stack is raised and the data element will be written to \( X \). The order of the element output can be clearly seen in the diagram below:

```
Row
1   2   3   4
a11 a12 a13 a14
a21 a22 a23 a24
a31 a32 a33 a34
```

Input

none

Example

The following program is a simple way to recall the elements of a matrix:

```
01*LBL "COU T"
02 "R010"
03 CLX
04 IJ=A
05*LBL 01
06 C>+
07 PSE
08 GTO 01
09 END
```
Further Hints

When the last element of a matrix has been output, the pointer will be positioned to a nonexistent element. Attempts to recall further elements will generate the error message "END OF FL".

Related Functions

>C+, >R+, C>-, R>+, R>-
The function C>- allows us to recall data array elements column by column, i.e. however, an array with i rows and j columns will the value of i reduced by 1 after each execution of the function. The stack is raised and the array element is written to X. The order of the element output can be seen clearly in the diagram below:

```
01*LBL "CBO
02 "R010"
03 DIM
04 IJ=A
05*LBL 01
06 C>-
07 PSE
08 GTO 01
09 END
```

Input
none

Example
The following program shows a simple way to output the elements of a matrix column by column, but in reverse order.
Further Hints

When the last element of a matrix has been output, the pointer will be positioned to a nonexistent element (in this case, the element 0.000). Attempts to recall further elements will result in the error message "END OF FL".

Related Functions

>C+, >R+, C>+, R>+, R>-
The function \texttt{R>\texttt{+}} enables us to output data array elements row by row, i.e. in an array with \textit{i} rows and \textit{j} columns, the value of \textit{j} will be incremented by 1 after the element is output. The stack is be raised and the array element is written to \texttt{X}. The order of the element output is shown clearly in the diagram below:

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
\texttt{Column} & \texttt{Row} \\
\hline
\texttt{a}_{11} & \texttt{a}_{12} & \texttt{a}_{13} & \texttt{a}_{14} \\
\hline
\texttt{a}_{21} & \texttt{a}_{22} & \texttt{a}_{23} & \texttt{a}_{24} \\
\hline
\texttt{a}_{31} & \texttt{a}_{32} & \texttt{a}_{33} & \texttt{a}_{34} \\
\hline
\end{tabular}
\end{center}

Input

none

Example

The following program shows how to output the elements of a matrix row by row:

\begin{verbatim}
01*LBL "CBO UT"
02 "R010"
03 CLX
04 IJ=A
05*LBL 01
06 R>+
07 PSE
08 GTO 01
09 END
\end{verbatim}
Further Hints

When the last element of a matrix has been output, the pointer will be positioned to a nonexistent element. Attempts to further recall elements will generate the error message "END OF FL".

Related Functions

>\texttt{C+}, \texttt{R+}, \texttt{C>}, \texttt{C>}, \texttt{R>}
The function \( R>\) allows us to output data array elements row by row, i.e. for an array with \( i \) rows and \( j \) columns, the value of \( j \) will be decremented by 1 after each element is output. The stack is raised and the array element is written to X. The order of the element output is clearly shown in the diagram below:

\[
\begin{array}{cccc}
| a_{11} | a_{12} | a_{13} | a_{14} | \\
| a_{21} | a_{22} | a_{23} | a_{24} | \\
| a_{31} | a_{32} | a_{33} | a_{34} | \\
\end{array}
\]

Input
none

Example
The following program shows how to recall the elements of a matrix backwards, row by row

\[
\begin{align*}
01 & \text{LBL "RBO UT")} \\
02 & "R010" \\
03 & \text{DIM} \\
04 & \text{IJ=A} \\
05 & \text{LBL 01} \\
06 & R>\ \\
07 & \text{PSE} \\
08 & \text{GTO 01} \\
09 & \text{END}
\end{align*}
\]
Further Hints

When the last element of a matrix has been output, the pointer will be positioned to a nonexistent element. Attempts to recall further elements will generate the error message "END OF FL".

Related Functions

> C+, > R+, C>+, C>-, R>
Functions for Shifting and Exchanging Elements

C<>C

The function C<>C exchanges column $kkk$ with column $lll$ of the specified data array.

**Input**

X register : $kkk.lll$

ALPHA register : array name

**Example**

We want to exchange the first column of the data array shown below with the second column; the original data array is:

<table>
<thead>
<tr>
<th>Row</th>
<th>$a_{11}$</th>
<th>$a_{12}$</th>
<th>$a_{13}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>$a_{21}$</td>
<td>$a_{22}$</td>
<td>$a_{23}$</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
After the sequence 2.001 \texttt{C<>C} we get the following array:

\begin{tabular}{|c|c|c|}
  \hline
  \text{Column} & $a_{11}$ & $a_{12}$ & $a_{13}$ \\
  \hline
  \text{Row} & 2 & 1 & 3 \\
  \hline
  & 5 & 4 & 6 \\
  \hline
\end{tabular}

\textbf{Related Functions}

\texttt{R<>R, MOVE, SWAP}
**R<>R**

The function **R<>R** exchanges row *kkk* with row *lll* of the specified data array.

**Input**

- **X register**: *kkk.lll*
- **ALPHA register**: array name

**Example**

We want to exchange the first row of the data shown below with the second row; the original data array is:

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><em>a</em>&lt;sub&gt;11&lt;/sub&gt;</td>
<td><em>a</em>&lt;sub&gt;12&lt;/sub&gt;</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><em>a</em>&lt;sub&gt;21&lt;/sub&gt;</td>
<td><em>a</em>&lt;sub&gt;22&lt;/sub&gt;</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

After the sequence 2.001 **R<>R** the following array is left:

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><em>a</em>&lt;sub&gt;11&lt;/sub&gt;</td>
<td><em>a</em>&lt;sub&gt;12&lt;/sub&gt;</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td><em>a</em>&lt;sub&gt;21&lt;/sub&gt;</td>
<td><em>a</em>&lt;sub&gt;22&lt;/sub&gt;</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Further Hints

The exchanging of rows occurs when pivoting a matrix.

Related Functions

C<>C, MOVE, SWAP
MOVE

The function MOVE copies the specified elements from the OPI array to the specified elements in the RES array. Only "rectangular" parts of a matrix can be moved; the block is determined by the description of two opposing corner elements iii.jjj and kkk.lli of the OPI matrix. The upper left corner, mmm.mnn of the goal block must be specified in the Z register.

Input

X register : iii.jjj
Y register : kkk.lli
Z register : mmm.mnn
ALPHA register : OPI1,RES

Example

We wish to transfer the elements a_{32}, a_{33}, a_{42} and a_{43} of the illustrated 4*3 data array "A" into data array "B". The goal elements in array "B" shall be b_{22}, b_{23}, b_{32} and b_{33}. The data arrays "A" and "B" are shown below:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Column</td>
<td>Column</td>
</tr>
<tr>
<td>Row</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a_{11}</td>
</tr>
<tr>
<td>1</td>
<td>a_{12}</td>
</tr>
<tr>
<td>2</td>
<td>a_{13}</td>
</tr>
<tr>
<td>a_{21}</td>
<td>a_{22}</td>
</tr>
<tr>
<td>4</td>
<td>a_{23}</td>
</tr>
<tr>
<td>a_{31}</td>
<td>a_{32}</td>
</tr>
<tr>
<td>7</td>
<td>a_{33}</td>
</tr>
<tr>
<td>a_{41}</td>
<td>a_{42}</td>
</tr>
<tr>
<td>10</td>
<td>a_{43}</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Column</td>
<td>Column</td>
</tr>
<tr>
<td>Row</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b_{11}</td>
</tr>
<tr>
<td>0</td>
<td>b_{12}</td>
</tr>
<tr>
<td>0</td>
<td>b_{13}</td>
</tr>
<tr>
<td>b_{21}</td>
<td>b_{22}</td>
</tr>
<tr>
<td>0</td>
<td>b_{23}</td>
</tr>
<tr>
<td>b_{31}</td>
<td>b_{32}</td>
</tr>
<tr>
<td>0</td>
<td>b_{33}</td>
</tr>
<tr>
<td>b_{41}</td>
<td>b_{42}</td>
</tr>
<tr>
<td>0</td>
<td>b_{43}</td>
</tr>
</tbody>
</table>

(C) Copyright W&W Software Products GmbH 1985 4.35
Naturally the data arrays may have different dimensions. After the sequence `ALPHA A,B ALPHA 2.002 ENTER 4.003 ENTER 3.002 MOVE` the following picture of the data arrays "A" and "B" develops:

<table>
<thead>
<tr>
<th></th>
<th>Column</th>
<th></th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a_{11}</td>
<td>a_{12}</td>
<td>a_{13}</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>a_{21}</td>
<td>a_{22}</td>
<td>a_{23}</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>a_{31}</td>
<td>a_{32}</td>
<td>a_{33}</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>a_{41}</td>
<td>a_{42}</td>
<td>a_{43}</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>b_{11}</td>
<td>b_{12}</td>
<td>b_{13}</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>b_{21}</td>
<td>b_{22}</td>
<td>b_{23}</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>b_{31}</td>
<td>b_{32}</td>
<td>b_{33}</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>b_{41}</td>
<td>b_{42}</td>
<td>b_{43}</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The array "A" was not changed in any way, in array "B" all elements are in their desired places.

**Further Hints**

The block that is to be moved can be described in several ways:

1) the upper left and the lower right element or
2) the lower left and the upper right element

The contents of the X and Y registers may be exchanged, i.e. `kkk.lll` may be placed in X and `iii.jjj` may be placed in Y.

**Related Functions**

`C<>C, R<>R, SWAP`
SWAP

The function SWAP exchanges selected elements of a specified data array with the elements of a different data array. Only "rectangular" parts of a matrix may be exchanged. The block is determined by the description of two opposing corner elements $i_{ii}, j_{jj}$ and $k_{kk}, l_{ll}$ (much like MOVE). The upper left corner of the second data array, $mmm.nnn$ must be placed in Z.

Input

X register : $i_{ii}, j_{jj}$
Y register : $k_{kk}, l_{ll}$
Z register : $mmm.nnn$
ALPHA register : OP1, OP2

Example

We want to exchange the elements $a_{32}, a_{33}, a_{42}$ and $a_{43}$ of the illustrated $4 \times 3$ data array "A" with the elements $b_{22}, b_{23}, b_{32}$ and $b_{33}$ of data array "B". Data arrays "A" and "B" are depicted below:

<table>
<thead>
<tr>
<th>Matrix A</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row</td>
<td>$a_{11}$</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$a_{21}$</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$a_{31}$</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>$a_{41}$</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Matrix B</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row</td>
<td>$b_{11}$</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$b_{21}$</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$b_{31}$</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$b_{41}$</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

(C) Copyright W&W Software Products GmbH 1985 4.37
Naturally the data arrays may have different dimensions. After the sequence **ALPHA A,B ALPHA 2.002 ENTER 4.003 ENTER 3.002 SWAP** the following picture of data arrays "A" and "B" develops:

The selected elements of arrays "A" and "B" have been exchanged.

**Further Hints**

The block to be moved can be described in several different ways:

1) the upper left and the lower right element or
2) the lower left and the upper right element

Moreover the X and Y registers may be exchanged, i.e. **kkk.ill** may be placed in X and **iii.jjj** may be put into Y.

**Related Functions**

*C<>C, R<>R, MOVE*
Functions for Determining the Extreme Values of Array Elements

These functions are needed for numeric solution procedures and Algorithms.

MAX

The function MAX sets the pointer to the greatest element of the given data array. This element is placed in the X register.

Input

ALPHA register : OPI

Example

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a_{11}$</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$a_{21}$</td>
</tr>
<tr>
<td>$0$</td>
<td>-4</td>
</tr>
<tr>
<td></td>
<td>$a_{31}$</td>
</tr>
<tr>
<td>$0$</td>
<td>-8</td>
</tr>
<tr>
<td></td>
<td>$a_{41}$</td>
</tr>
<tr>
<td>$0$</td>
<td>-12</td>
</tr>
</tbody>
</table>

When using the function MAX on the above depicted data array, the pointer is set to element $a_{14}$ and a value of 4 is output to X.
Further Hints

The function MAX outputs the largest element, and not, contrary to MAXAB, the greatest absolute element. If there is no element greater than element $a_{11}$ (for example all array elements equal 0), then it is output. In this case the pointer will be positioned on element $a_{11}$ as well.

Related Functions

CMAXAB, MAX, MAXAB, MIN, PIV, R>R?, RMAXAB
MAXAB

The function MAXAB sets the pointer to the element with the largest absolute value in the given data array. The absolute value of this element is placed in X.

Input

ALPHA register : OPI

Example

<table>
<thead>
<tr>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{11}$</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>$a_{21}$</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>$a_{31}$</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>$a_{41}$</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

The greatest absolute element of the depicted array is -36. The function MAXAB sets the pointer on the element $a_{44}$ and gives its absolute value (thus 36) in the X register.
Further Hints

The element that is output to the X register is only equal to the element in the given position when this element is positive. If the element is negative, its absolute value will be placed in the X register. If there is no element whose absolute value is greater than element $a_{1l}$ (for example all array elements equal 0), it will be output. The pointer will be positioned on element $a_{1l}$ as well.

Related Functions

CMAXAB, MAX, MIN, PIV, R>R?, RMAXAB
CMAXAB

The function CMAXAB sets the pointer to the greatest absolute element of column $j\,jj$ of the given data array. This element is output in the X register.

Input

X register : $jj$  
ALPHA register : $OP1$

Example

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a_{11}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

After the key sequence 3 CMAXAB the pointer is positioned to element $a_{43}$ and its absolute value (24) is placed into X.

Further Hints

The search for the greatest absolute element occurs only in the given column. If there is no element of column $j$ whose absolute value is greater than element $a_{jj}$ (for example all column elements equal 0), it is output. The pointer is positioned to element $a_{1j}$ as well.

Related Functions

CMAXAB, MAX, MIN, PIV, R>R?, RMAXAB
RMAXAB

The function RMAXAB sets the pointer to the position of the element with the greatest absolute value in the specified row $iii$ of the given data array.

Input

X register : $iii$

ALPHA register : OP1

Example

<table>
<thead>
<tr>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{11}$</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>$a_{21}$</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>$a_{31}$</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>$a_{41}$</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

After the key sequence 2 RMAXAB the pointer is set to element $a_{24}$ and the number 12 can be seen in the X register.

Further Hints

The search for the greatest absolute element takes place only in the given row. If in row $i$ there is no element of greater absolute value than element $a_{ij}$ (for example all row elements equal 0), it is output. The pointer is positioned to this element $a_{ij}$ as well.

Related Functions

CMAXAB, MAX, MAXAB, MIN, PIV, R<R?
The function **MIN** sets the pointer to the smallest element of the specified data array. This element is output to the X register.

**Input**

ALPHA register : OPI

**Example**

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_{11}</td>
<td>a_{12}</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>a_{21}</td>
<td>a_{22}</td>
</tr>
<tr>
<td>0</td>
<td>-4</td>
</tr>
<tr>
<td>a_{31}</td>
<td>a_{32}</td>
</tr>
<tr>
<td>0</td>
<td>-8</td>
</tr>
<tr>
<td>a_{41}</td>
<td>a_{42}</td>
</tr>
<tr>
<td>0</td>
<td>-12</td>
</tr>
</tbody>
</table>

After executing the function **MIN** the pointer is set to a_{44} and the value -36 is placed into X.

**Further Hints**

If a_{jj} is the smallest element of the array (for example all array elements equal 0), it is output. The pointer is positioned to this element a_{jj} as well.

**Related Functions**

CMAXAB, MAX, MAXAB, PIV, R>R?, RMAXAB
PIV

The function PIV sets the element pointer to the coordinates of the element, under the principal diagonal of the selected column $jj$ of the specified array, with the greatest absolute value. The absolute value of this element is placed in X.

Input

X register : $jj$
ALPHA register : OP1

Example

<table>
<thead>
<tr>
<th>Row</th>
<th>$a_{11}$</th>
<th>$a_{12}$</th>
<th>$a_{13}$</th>
<th>$a_{14}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>-12</td>
<td>-24</td>
<td>-36</td>
</tr>
</tbody>
</table>

Column

With the key sequence 2 PIV the pointer is set to element $a_{42}$ and the absolute value of this element (12) is put into the X register.

Further Hints

The function PIV uses only the integer part of the X register. The fractional part is ignored during the procedure. The search for the greatest absolute element starts with element $a_{jj}$ (on the main diagonal) and is continued downward in column $j$.

Related Functions

CMAXAB, MAX, MAXAB, MIN, R>R?, RMAXAB
R>R?

The function R>R? compares the elements of row $kkk$ with those of row $lll$ of the specified data array. The elements are compared columnwise, starting with column one. If the elements of this column are the same, the two elements of the next column are compared until there are either two unequal elements or until the end of the row is reached. The answer is "YES", as soon as an element of the row $lll$ is greater than the element in the same column of row $kkk$. When finding an element in row $lll$ which is smaller than the corresponding element in row $kkk$, the answer is "NO" and a step is skipped in a running program.

Input

X register : $kkk.lll$
ALPHA register : $OP1$

Example

We want to check if the first row of the depicted data array is greater than the third:

\[
\begin{array}{cccc}
\alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\
1 & 2 & 3 & 4 \\
\alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\
5 & 6 & 7 & 8 \\
\alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\
9 & -8 & -16 & -24 \\
\alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \\
13 & -12 & -24 & -36 \\
\end{array}
\]

(C) Copyright W & W Software Products GmbH 1985
After keying in the sequence 1.003 \texttt{R>R?} we get the answer "NO", since the element in row 3 of the first column is greater than that in row 1.

**Further Hints**

In a running program if the answer is "YES", the next program line is executed, otherwise a line is skipped.

**Related Functions**

\texttt{MAX, MAXAB, MIN, CMAXAB, RMAXAB, PIV}
Sums and Norms

These functions are, like the functions for finding the extreme values, needed for numeric solution methods and algorithms. Using norms it can be established whether a linear equation is singular, i.e. whether it is definitely solvable or not. Moreover, norms can give information about the reliability of a result; in this case we talk about the conditional number of the matrix.

SUM

SUM adds all elements of the specified data array and writes the result in X.

Input

ALPHA register: OPI

Example

```
Column

<table>
<thead>
<tr>
<th>a_{11}</th>
<th>a_{12}</th>
<th>a_{13}</th>
<th>a_{14}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>a_{21}</td>
<td>a_{22}</td>
<td>a_{23}</td>
<td>a_{24}</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>a_{31}</td>
<td>a_{32}</td>
<td>a_{33}</td>
<td>a_{34}</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>a_{41}</td>
<td>a_{42}</td>
<td>a_{43}</td>
<td>a_{44}</td>
</tr>
<tr>
<td>0</td>
<td>-12</td>
<td>-24</td>
<td>-36</td>
</tr>
</tbody>
</table>
```
After keying the sequence ALPHA A ALPHA SUM the sum of all array elements (6), is put into the X register.

Further Hints

The function SUM does not have any influence on the pointer, therefore its position is not changed.

Related Functions

SUMAB, CSUM, RSUM
SUMAB

**SUMAB** adds the absolute values of all elements of the specified data array and writes the result to X.

**Input**

ALPHA register: *OP1*

**Example**

```
<table>
<thead>
<tr>
<th>a_{11}</th>
<th>a_{12}</th>
<th>a_{13}</th>
<th>a_{14}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>a_{21}</td>
<td>a_{22}</td>
<td>a_{23}</td>
<td>a_{24}</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>a_{31}</td>
<td>a_{32}</td>
<td>a_{33}</td>
<td>a_{34}</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>a_{41}</td>
<td>a_{42}</td>
<td>a_{43}</td>
<td>a_{44}</td>
</tr>
<tr>
<td>0</td>
<td>-12</td>
<td>-24</td>
<td>-36</td>
</tr>
</tbody>
</table>
```

After the sequence ALPHA A ALPHA SUMAB the sum of the absolute values of all the array elements (150), is placed into X.

**Further Hints**

The function **SUMAB** does not have any influence on the position of the pointer, so its position remains the same.

**Related Functions**

SUM, CSUM, RSUM
CSUM

CSUM adds all elements of each column of the OPI data array and places the results into the RES data array.

Input

ALPHA register: OPI,RES

Example

To form the column sums of the depicted data array "A" (2 rows and 3 columns), one more data array is needed, which must also have 3 columns, but need only have one row. After the key sequence ALPHA A,B ALPHA CSUM the data array "B" contains the following elements:

<table>
<thead>
<tr>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>a11</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>a21</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

Further Hints

The only inputs needed prior to the execution of CSUM are the OPI and the RES matrices. The RES matrix must have the same number of columns as OPI but needs only one row.

Related Functions

SUM, SUMAB, RSUM
RSUM

RSUM adds all elements of each row of the given data array and writes the results in the second given data array.

Input

ALPHA register: OPI,RES

Example

To form the row sums of the depicted data array "A" (2 rows and 3 columns) a second data array is needed, which must also have 2 rows, but need only have one column. After keying the sequence ALPHA A,B ALPHA RSUM the data array "B" contains the respective row sums:

<table>
<thead>
<tr>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Further Hints

To calculate the results, RSUM always needs a data array which has the same number of rows as the data array whose row sums are to be calculated.

Related Functions

SUM, SUMAB, CSUM
The function **CNRM** computes the column norm, defined as the largest column sum of absolute values for a matrix, for the array specified in ALPHA and places the result in X. This norm is computed using the equation

\[ \| A \|_C = \max_{1 < j < n} \sum_{i=1}^{m} |a_{ij}| \]

**Input**

**ALPHA register** : *OP1*

**Example**

<table>
<thead>
<tr>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_{11} )</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>( a_{21} )</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>( a_{31} )</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>( a_{41} )</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

The key sequence **ALPHA OP1 ALPHA CNRM** yields the value 60 in the X register; this is the sum of the absolute values of the elements in the fourth column, which is the largest sum of absolute values of the array.

**Related Functions**

RNRM, FNRM
RNRM

The function RNRM calculates row norm, defined as the largest row sum of absolute values for a matrix, for the array specified in ALPHA. The result is written to the X register. The formula for this is:

\[ \| A \|_R = \max_{1 < i < m} \sum_{j=1}^{n} |a_{ij}| \]

Input

ALPHA register : OPI

Example

Using the key sequence ALPHA A ALPHA RSUM the sum of the absolute values of the elements in the fourth row, which is the largest row sum of absolute values of the depicted array. This sum (in this case 72) is put into the X register.

Related Functions

CNRM, FNRM
The function \textsc{fnrm} calculates the frobenius norm of the specified data array and writes the result to the X register. The formula for this is:

\[
\| A \|_F = \left( \sum a_{ij}^2 \right)^{1/2}
\]

**Input**

\textsc{alpha} register : \textit{op1}

**Example**

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_{11})</td>
<td>(a_{12})</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(a_{21})</td>
<td>(a_{22})</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>(a_{31})</td>
<td>(a_{32})</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>(a_{41})</td>
<td>(a_{42})</td>
</tr>
<tr>
<td>0</td>
<td>-12</td>
</tr>
</tbody>
</table>

After the key sequence \textsc{alpha} \textit{op1} \textsc{alpha} \textsc{fnrm} we get the value 51.6333 in the X register. This number is the calculated value of the frobenius norm of the depicted data array.

**Further Hints**

The frobenius norm of a square matrix corresponds to its euclidian length.

**Related Functions**

\textsc{cnrm}, \textsc{rnrm}
Functions for Mathematical Manipulation of Arrays

This section of routines for manipulating arrays includes eight functions with two subgroups of related functions. Included in the first of these groups are the functions M+, M-, M* and M/ which are very versatile, since many different possibilities for the manipulation of data arrays can be exploited by the judicious selection of operands. The execution time, compared to user code programs, is extremely short.

M+

The function M+ adds the elements with the same indices of two data arrays (only numerical data is allowed). The formula for this is:

\[ c_{ij} = a_{ij} + b_{ij} \]

Input

ALPHA register : OP1,OP2,RES

Example

We want to add the depicted arrays "A" and "B":

<table>
<thead>
<tr>
<th>Matrix A</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row</td>
<td>( a_{11} )</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>( a_{21} )</td>
<td>( a_{22} )</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Matrix B</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row</td>
<td>( b_{11} )</td>
</tr>
<tr>
<td>50</td>
<td>-47</td>
</tr>
<tr>
<td>( b_{21} )</td>
<td>( b_{22} )</td>
</tr>
<tr>
<td>20</td>
<td>7</td>
</tr>
</tbody>
</table>
Following the key sequence \texttt{ALPHA A,B,C \texttt{ALPHA M+}} the array "C" contains the following elements:

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
\text{Row} & \text{Column} & \\
\hline
\text{c}_{11} & \text{c}_{12} & \text{c}_{13} \\
51 & -45 & 12 \\
\hline
\text{c}_{21} & \text{c}_{22} & \text{c}_{23} \\
24 & 12 & 18 \\
\hline
\end{tabular}
\end{center}

\textbf{Further Hints}

If only \texttt{OPI} is mentioned in the \texttt{ALPHA} register, the value of each element in the specified data array is doubled. If, while executing the function, the error message "OUT OF RANGE" shows up, it signifies that parts of the data array have already been worked on, meaning that the result can not be used. If one of the operands is called "X", the value in the \texttt{X} register is supposed to be added to each element. The given data arrays must have the same dimensions.

\textbf{Related Functions}

\texttt{M-, M*, M/}
The function \( M^- \) subtracts elements with identical indices of two data arrays (only numeric data is allowed). The formula for this is:

\[
c_{ij} = a_{ij} - b_{ij}
\]

**Input**

ALPHA register : \( OP1, OP2, RES \)

**Example**

We want to subtract the depicted array "B" from array "A":

<table>
<thead>
<tr>
<th>Column</th>
<th>Matrix A</th>
<th>Matrix B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( a_{11} )</td>
<td>( b_{11} )</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>-47</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Column</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( a_{21} )</td>
<td>( b_{21} )</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>
After the key sequence **ALPHA A,B,C ALPHA M-** the array "C" contains the following elements:

Matrix C

<table>
<thead>
<tr>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_{11}$</td>
</tr>
<tr>
<td>-49</td>
</tr>
<tr>
<td>$c_{21}$</td>
</tr>
<tr>
<td>-16</td>
</tr>
</tbody>
</table>

**Further Hints**

If only *OP1I* is specified in ALPHA, then the value of each element will be 0. If during the execution of this function the error message "OUT OF RANGE" is generated, then parts of the data array have already been worked on, and the result cannot be used. If one of the operands is "X" the value in the X register is subtracted from each element. The given data arrays must have the same dimensions.

**Related Functions**

M+, M*, M/
**M***

The function **M*** multiplies elements of two data arrays with the same indices (only numerical data is allowed). The formula for this is:

\[ c_{ij} = a_{ij} \cdot b_{ij} \]

**Input**

ALPHA register: \textit{OP1,OP2,RES}

**Example**

We want to multiply the elements of the depicted arrays "A" and "B":

\[
\begin{array}{c|c|c}
\text{Matrix A} & \text{Matrix B} \\
\hline
\text{Column} & \text{Column} \\
\hline
a_{11} & b_{11} & 1 & 50 \\
a_{12} & b_{12} & 2 & -47 \\
a_{13} & b_{13} & 3 & 9 \\
\hline
a_{21} & b_{21} & 4 & 20 \\
a_{22} & b_{22} & 5 & 7 \\
a_{23} & b_{23} & 6 & 12 \\
\end{array}
\]
After the key sequence ALPHA $A,B,C$ ALPHA $M^*$ array "C" contains the following elements:

Matrix C

<table>
<thead>
<tr>
<th></th>
<th>$c_{11}$</th>
<th>$c_{12}$</th>
<th>$c_{13}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 1</td>
<td>50</td>
<td>-94</td>
<td>27</td>
</tr>
<tr>
<td>Row 2</td>
<td>80</td>
<td>35</td>
<td>72</td>
</tr>
</tbody>
</table>

Further Hints

The squaring of the elements of a data array is accomplished if only $OPI$ is placed in the ALPHA register. If, during execution of this function, the error message "OUT OF RANGE" is displayed, then parts of the data array have already been worked on, and the result can not be used. The given data arrays must have the same dimensions, if one of the operands is "X" then we want to multiply each element by the value in the X register (scalar multiplication).

Related Functions

M+, M-, M/
The function \( M/ \) divides the elements of two data arrays with the same indices (only numeric data is allowed). The formula for this is:

\[
c_{ij} = \frac{a_{ij}}{b_{ij}}
\]

Input

ALPHA register : \( OP1, OP2, RES \)

Example

To prepare this example we want to transfer the elements of array "C" into array "A": ALPHA C,A ALPHA 1.001 ENTER 2.003 ENTER MOVE. After doing this the arrays "A" and "B" contain the following elements:

Matrix A

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( a_{11} )</td>
<td>-94</td>
</tr>
<tr>
<td>( a_{21} )</td>
<td>50</td>
<td>80</td>
</tr>
<tr>
<td>( a_{12} )</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>( a_{13} )</td>
<td>72</td>
<td></td>
</tr>
</tbody>
</table>

Matrix B

<table>
<thead>
<tr>
<th>Column</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( b_{11} )</td>
<td>50</td>
</tr>
<tr>
<td>( b_{21} )</td>
<td>20</td>
</tr>
<tr>
<td>( b_{12} )</td>
<td>20</td>
</tr>
<tr>
<td>( b_{22} )</td>
<td></td>
</tr>
<tr>
<td>( b_{13} )</td>
<td></td>
</tr>
<tr>
<td>( b_{23} )</td>
<td>12</td>
</tr>
</tbody>
</table>
With the help of key sequence **ALPHA A,B,C ALPHA M/** we now divide the elements of array "A" by the elements of array "B". Now array "C" contains the following elements:

![Matrix C](image)

**Further Hints**

If only **OP1** is mentioned in **ALPHA**, and if none of the elements in the matrix are zero, then a unit matrix is obtained (all elements equal to 1). If, during execution of this function, the error message "OUT OF RANGE" shows up, it signifies that parts of the data array have already been worked on, meaning the result can not be used. The given data arrays must have the same dimensions, unless one of the operands is "X" in which case the value of the X register is used.

**Related Functions**

M+, M-, M*
The function \( M*M \) allows the multiplication of matrices. This multiplication is done according to the following rule:

\[
c_{ik} = \sum_{j=1}^{n} a_{ij} \cdot b_{jk}
\]

**Input**

**ALPHA register** : \( OP1, OP2, RES \)

**Example**

Using the given matrix "A" and the given column vector "B" and the key sequence \( \text{ALPHA A,B,C ALPHA } M*M \) we get the resultant column vector "C" of a series of equations with a unique solution. For this application, the dimensions of the matrices must be linked, so that if "A" is a square 4 * 4 matrix then any operand "B" must have four rows. Since in this case "B" is a column vector, this array has only one column. The resultant array is then an array of the dimension 4 * 1. The arrays "A", "B", and "C" and their elements are shown below:

\[
\begin{array}{cccc}
\text{Row} & a_{11} & a_{12} & a_{13} & a_{14} \\
1 & 1 & 2 & 3 & 4 \\
2 & 5 & 6 & 7 & 8 \\
3 & 9 & 10 & 11 & 12 \\
4 & 13 & 14 & 15 & 16 \\
\end{array}
\quad
\begin{array}{c}
\text{Column} \\
b_{11} \\
n & -3 \\
5 & 8 \\
9 & 11 \\
13 & 16 \\
\end{array}
\quad
\begin{array}{c}
c_{11} \\
18 \\
38 \\
58 \\
78 \\
\end{array}
\]
Further Hints

The dimensions of the given data arrays OP1 and OP2 must be linked i.e. if the array OP1 has the dimension i* j, array OP2 must have the dimension j* k. If the dimensions of the operands are not related in this manner then the error message "DIM ERR" appears. The result of this operation is a data array of the dimension i* k, meaning the array RES must already exist with this dimension before executing this function.

An important application of M* M lies in the solution of non singular, linear equation systems: if an inverse (square matrix which has dimensions i*i) matrix is multiplied with a column vector, the result will also be a vector. This operation is especially rewarding if one fixed array is multiplied by a number of different column vectors (for example, for the analysis of a series of many measurements with a single system). In this case it is not necessary to solve the equation system anew every time, it suffices to execute the function M* M with successive column vector.

Related Functions

M+, M-, M*, M/
**YC+C**

The function YC+C makes it possible to add a multiple (factor in Y) of column $i$ to column $j$.

**Input**

- **ALPHA register**: $OP1$
- **X register**: $iii.jjj$
- **Y register**: factor

**Examples**

We want to work on the given data array "A":

```
Column
Row   a_{11}  a_{12}  a_{13}
     1      2      3
      a_{21}  a_{22}  a_{23}
     4      5      6
```

The key sequence **ALPHA A ALPHA 5 ENTER 1.001 YC+C** yields the array:

```
Column
Row   a_{11}  a_{12}  a_{13}
     6      2      3
      a_{21}  a_{22}  a_{23}
     24     5      6
```

The first column was multiplied by 5 and added to itself.
In the second example we shall also start with array "A":

<table>
<thead>
<tr>
<th></th>
<th>(a_{11})</th>
<th>(a_{12})</th>
<th>(a_{13})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>(a_{21})</td>
<td>(a_{22})</td>
<td>(a_{23})</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

After the key sequence `ALPHA A ALPHA -50 ENTER 3.001 YC+C` the following picture develops:

<table>
<thead>
<tr>
<th></th>
<th>(a_{11})</th>
<th>(a_{12})</th>
<th>(a_{13})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row</td>
<td>(-149)</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>(a_{21})</td>
<td>(a_{22})</td>
<td>(a_{23})</td>
</tr>
<tr>
<td></td>
<td>(-296)</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

The elements of the third column were multiplied by -50 and then added to the elements of the first column.

Further Hints

If, during execution of this function, the error "OUT OF RANGE" is displayed, then parts of the data array have been worked on, and the result is useless.
R-QR

The function R-QR performs one step of a Gauss procedure, which transforms an array so that all the elements under the principal diagonal become equal to zero. Using this method a system of equations may be solved by backwards insertion. Execution of the function proceeds according to the following protocol:

\[
\text{for } Q = a_{kl} / a_{ll} \text{ we obtain:}
\]

\[
\text{for } 1 < j < n : a_{kj} = a_{kj} - Q \cdot a_{lj}
\]

The element coordinates \textit{kkk.ill} are of course specified as input. After execution, the element \(a_{kl}\) is transformed to zero. Thus \textit{ill} specifies the row which, after being multiplied by \(Q\), is subtracted from row \textit{kkk}; so \(Q = a_{kl}/a_{ll}\). Note that for the matrix upon which this operation is performed, no diagonal element may equal zero.

Input

\textbf{X register : kkk.ill}

Example

As an example we will work on the equation system \(A \times x = 0\), "A" is a \(4 \times 4\) matrix and is shown on the next page.
<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>( a_{11} )</th>
<th>( a_{12} )</th>
<th>( a_{13} )</th>
<th>( a_{14} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_{21} )</td>
<td>( a_{22} )</td>
<td>( a_{23} )</td>
<td>( a_{24} )</td>
<td>( a_{31} )</td>
<td>( a_{32} )</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

using 4.001 R-QR we get:

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>( a_{11} )</th>
<th>( a_{12} )</th>
<th>( a_{13} )</th>
<th>( a_{14} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_{21} )</td>
<td>( a_{22} )</td>
<td>( a_{23} )</td>
<td>( a_{24} )</td>
<td>( a_{31} )</td>
<td>( a_{32} )</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>( \theta )</td>
<td>(-12)</td>
</tr>
</tbody>
</table>

with 3.001 R-QR we get:

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>( a_{11} )</th>
<th>( a_{12} )</th>
<th>( a_{13} )</th>
<th>( a_{14} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_{21} )</td>
<td>( a_{22} )</td>
<td>( a_{23} )</td>
<td>( a_{24} )</td>
<td>( a_{31} )</td>
<td>( a_{32} )</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>( \theta )</td>
<td>(-12)</td>
</tr>
</tbody>
</table>

(C) Copyright W&W Software Products GmbH 1985
with 2.001 R-QR we get:

<table>
<thead>
<tr>
<th>Column</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a_{11}</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>a_{21}</td>
<td>0</td>
<td>-4</td>
<td>-8</td>
</tr>
<tr>
<td>a_{31}</td>
<td>0</td>
<td>-8</td>
<td>-16</td>
</tr>
<tr>
<td>a_{41}</td>
<td>0</td>
<td>-12</td>
<td>-24</td>
</tr>
</tbody>
</table>

with 4.002 R-QR we get:

<table>
<thead>
<tr>
<th>Column</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a_{11}</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>a_{21}</td>
<td>0</td>
<td>-4</td>
<td>-8</td>
</tr>
<tr>
<td>a_{31}</td>
<td>0</td>
<td>-8</td>
<td>-16</td>
</tr>
<tr>
<td>a_{41}</td>
<td>0</td>
<td>-12</td>
<td>-24</td>
</tr>
</tbody>
</table>

with 3.002 R-QR we get:

<table>
<thead>
<tr>
<th>Column</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a_{11}</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>a_{21}</td>
<td>0</td>
<td>-4</td>
<td>-8</td>
</tr>
<tr>
<td>a_{31}</td>
<td>0</td>
<td>-8</td>
<td>-16</td>
</tr>
<tr>
<td>a_{41}</td>
<td>0</td>
<td>-12</td>
<td>-24</td>
</tr>
</tbody>
</table>
It is plain to see that by repeated use of **R-QR** results in the formation of an upper triangular array, as the shaded lower elements are successively transformed to zero. Note that in the course of transforming the array for this example we have used additional elements from the upper triangular portion of the array and these have also become zero. At each step, the element which must be transformed is shaded darker gray. An example of a system of inhomogeneous solutions of the $A \times x = b$ (with a matrix consisting of $A$ and the column vector $b$) is relinquished.

**Further Hints**

In contrast to the so called "L-R analysis" (see **R-PR**) no column vector may be introduced at a later stage of execution.

**Related Function**

**R-PR**
The function **R-PR** works similar to the function **Q-QR**. Since, when working with Gaussian algorithms, all elements of the lower triangle matrix become equal to 0. Thus it is possible to restore the original matrix by working backwards from the transformed array. Furthermore, it is possible to introduce a new column vector as part of the process. The algorithm works exactly the same way as in **R-QR**:

\[
\begin{align*}
\text{for } Q &= a_{kl} / a_{ll} \text{ we obtain:} \\
\text{for } l < j < n : \quad a_{kj} &= a_{kj} - Q \cdot a_{lj} \\
\text{for } j = l \quad a_{kl} &= Q \\
\text{for } 1 < j < l : \quad a_{kj} &= a_{kj}
\end{align*}
\]

For the elements \( l+1 \) to \( n \) of line \( k \) the same operation as in **R-QR** is used; the element \( a_{kl} \) gets the value \( Q \) and all elements to the left of \( a_{kl} \) stay the same. \( kkk.ill \) is placed in the X register. \( ill \) gives us the row that multiplied by \( Q \) and then is subtracted from row \( kkk \). Thus

\[
Q = a_{kl} / a_{ll}
\]

In this operation the elements may not be equal to 0!

**Input**

- **X register**: \( kkk.ill \)
- **ALPHA register**: array name
Example

As an example we will work on the equation system $A \cdot x = b$. "A" is a $4 \times 4$ matrix and the column vector "b" (inhomogenous part) is moved into the matrix, so that there arises a $4 \times 5$ matrix:

\[
\begin{array}{cccccc|c}
 & & & & & & b \\
 a_{11} & a_{12} & a_{13} & a_{14} & & 1 \\
a_{21} & a_{22} & a_{23} & a_{24} & & 2 \\
a_{31} & a_{32} & a_{33} & a_{34} & & 3 \\
a_{41} & a_{42} & a_{43} & a_{44} & & 4 \\
\end{array}
\]

Depicted above are matrix "A" and column vector "b", and below the new $4 \times 5$ matrix which has been combined with the column vector:

\[
\begin{array}{cccccc|c}
 & & & & & & b \\
 a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & 0 \\
a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & 1 \\
a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & 2 \\
a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & 3 \\
\end{array}
\]
using 4.001 R-PR we get:

<table>
<thead>
<tr>
<th>Row</th>
<th>(a_{11})</th>
<th>(a_{12})</th>
<th>(a_{13})</th>
<th>(a_{14})</th>
<th>(a_{15})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

Column

<table>
<thead>
<tr>
<th>Column</th>
<th>(a_{11})</th>
<th>(a_{12})</th>
<th>(a_{13})</th>
<th>(a_{14})</th>
<th>(a_{15})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

using 3.001 R-PR we get:

<table>
<thead>
<tr>
<th>Row</th>
<th>(a_{11})</th>
<th>(a_{12})</th>
<th>(a_{13})</th>
<th>(a_{14})</th>
<th>(a_{15})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

using 2.001 R-PR we get:

<table>
<thead>
<tr>
<th>Row</th>
<th>(a_{11})</th>
<th>(a_{12})</th>
<th>(a_{13})</th>
<th>(a_{14})</th>
<th>(a_{15})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>
using 4.002 R-PR we get:

\[
\begin{array}{cccccc}
 & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\
Row & 1 & 2 & 3 & 4 & 0 \\
 & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\
 & 5 & -4 & -8 & -12 & 1 \\
 & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\
 & 9 & 2 & 0 & 0 & 0 \\
 & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\
 & 13 & 3 & 0 & 0 & 0 \\
\end{array}
\]

using 3.002 R-PR we get:

\[
\begin{array}{cccccc}
 & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\
Row & 1 & 2 & 3 & 4 & 0 \\
 & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\
 & 5 & -4 & -8 & -12 & 1 \\
 & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\
 & 9 & -8 & -16 & -24 & 2 \\
 & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\
 & 13 & 3 & 0 & 0 & 0 \\
\end{array}
\]

Further Hints

The L-R analysis is practically the same as the Gauss procedure, except that here the moving in of a new column vector is possible, since the factors Q are saved. These factors Q are placed in the lower triangle matrix, whose elements would otherwise be equal to 0. For the backwards input though only the upper triangle matrix may be examined.

Related Function

R-QR
Program Examples

We shall now present two helpful programs that will show you the use of some CCD-Module matrix functions to demonstrate their use.

Gaussian-algorithm

The first program allows the solving of a linear equation system with a quadratic coefficient matrix by the Gauss elimination procedure. The program consists of several exchangable parts that need information in any of two registers. In this case the registers 00 and 05 were chosen.

Using the program:

Using the program subroutine "ABIN", all values of the expanded coefficient matrix are input. The transformation into a triangular form as well as the final output of the resultant vector are handled by the subroutine "TRANS".

```
01*LBL "ABI" Start of the input routine (R01 is used as N" the control register.)
02 "DIM(N)? Requests the input of the dimension n of the equation system
03 PROMPT and stores it in R00
04 STO 00
05 E Lines 5 - 10 generate the number nnn.mmm
06 + which is produced for the dimensioning of
07 E3 the matrix, where mmm = n+1
08 /
09 RCL 00
10 +
11 "NAME" Input of the matrix name (for example
12 PMTA "R006" for a matrix in the main memory)
16 maximum 6 letters
13 ASTO 05 Saves this name in R05
14 MDIM Creation of the matrix
15 STO 01 Storage of the control index for data input
16 CLX Using IJ=A, set pointer to a_A (not necessary when working with register matrices, see MDIM)
```
If wrong element input, there is a possibility of correction. Input in X: 

Selection of the chosen matrix (name in R05) as the current matrix

Start of the input routine

Has the last element already been read?

Yes, End of input loop

Production of message "A(" 

Appending of i: "A(i"

X: current pointer iii.jjj

Production of j

ALPHA: "A(i," 

Is j greater than the dimension n?

Yes, input of Y values. ALPHA: "Y(" 

Is j greater than dimension n?

Yes, in X j is substituted by iii.jjj

Appending of the integer value to ALPHA

ALPHA: "A(i,j):" or "B(i):"

Reading of the current pointer position

and the element on this position

Appending of this element to ALPHA

Obtain the current pointer position

Old element to X

Input of a new value or use

of the old as well as storing the value

Back to the start of the input routine

Start of the program for the transformation of the matrix to triangle shape

The matrix name must, for PIV and R-QR (or R-PR), be in ALPHA

All in all the pivoting is executed n-1 times
Outer routine \( i=1-(n-1) \)
Calculation of the pivot line number
\( Y \) runs from \( (n-1) \) to 1
\( i=n-Y \)
Calculate \( iii.iii \)
This \( E3 \) moves the \( Y \) contents, as after \( LBL \) 02 to \( T \).
Outer counter now in \( Z \)
Search for the pivot element
If this element is equal to 0, there exists
no one definite solution: Backtrack with \( X=0 \)
Pivot element is of no interest
If the pointer of the pivot element is unequal to the calculated one:
exchange lines
Put current pointer on \( iii.iii \)

Inner routine: \( j=n-i+1 \)
Calculate pointer \( jjj.iii \)
Pivot pointer
Counter from \( (n-1)-1 \)
Transformation of line \( j \) starting from column \( i \)
If counter is greater than 0
work on next highest row
Bring outer counter to \( X \)
If counter is greater than 0, work on next row under matrix
\( X \) not equal to 0 if definite solution (if called out as a subprogram)

Calculation of the resultant vector
\( ALPHA: \) matrix name
Production of \( 0,(nnn+1) \)
Use of register \( M \) as scratch register
\( X: nnn.(nnn+1) \)
89*LBL 04
90 INT
91 RCL X
92 E3
93 /
94 +
95*LBL 05
96 IJ=  
97 INT
98 RCL [ 
99 +
100 C>-
101 ?IJ
102 X<> Z
103 IJ=  
104 C>-
105 X<>Y
106 IJ=  
107 RDN
108 X<>Y
109 /
110 STO
111 >C+
112 RDN
113 X=0?
114 GTO 07
115*LBL 06
116 IJ=  
117 C>-
118 RCL \ 
119 *
120 ?IJ
121 X<> Z
122 INT
123 RCL [ 
124 +

Main routine: \(i = n - 1\)
Calculation of \(iii.iii\)

Put pointer on \(iii.iii\)
Get pointer of the result element to be calculated: \(iii.(nnn+1)\)

Put pointer on \(iii.(nnn+1)\)
Read element \(iii.iii\), decrement line pointer. This pointer is 0, if the upper line has been worked on see comparison \(X = 0?\)

Read element
Again put pointer on \(iii.(nnn+1)\)

Calculate result element \(x_i\)
\[x_i = a_i n + 1 / a_i n\]
Storing in help register N as well as at his result place \(iii.(nnn+1)\)

If first line has been worked on:
ready, output!

Inner routine: \(j = i - 1 - 1\)
Put pointer to \(jjj.iii\)
Product \(a_{j,i} * x_i\)

Read pointer \((jjj-1).iii\) and save
Calculation of \(jjj.(nnn+1)\)

(C) Copyright W&W Software Products GmbH 1985
Subtraction of product \(a_j,i \cdot x_i\) from current value \(x_i\) read \(a_j,n+1\)

Subtract product

Put pointer on \(jjj.(nnn+1)\)

Store difference
If pointer on \(a_j\), has not yet jumped to line \(n\) (by \( \text{C}_j \)), \(j\) has not yet reached 1
Line \(i-1\) is next to be worked on

Back to main routine

End of calculation and output of the result vector \((x_i\) values)

Matrix name in ALPHAProduce pointer \(1.(n+1)\)

Put pointer on \(x_i\)
Start of output routine

Set for message \("X(i)="\)

Reading of the value \(x_i\) and appending to ALPHAPrint, if printer present
158 PRA
159 FC?C 25
160 PROMPT Otherwise message of x; values
161 GTO 08 Back to start of output routine
162 END End of program

PLNG "ABIN"
301 BYTES
Calculation of the Inverse Matrix

The program "INV" calculates the inverse \( A^{-1} \) of a given quadratic matrix \( A \). After calculating the inverse the linear equation systems of the form \( A \cdot x = b \) are easily solved with the help of a simple matrix multiplication (for example using the function \( M \cdot M \) of the CCD-Module):

\[
x = A^{-1} \cdot b
\]

The program INV solves singular matrices as well. For this, INV calculates a conditional number that allows for a statement about the exactness of the solution to be made. The relative error of the solution \( x \) can now be determined by multiplying the conditional number with the relative error of the column vector \( b \) (thus on the HP-41 minimal \( 1 \times 10^{-10} \)).

Use:

A name of a previously arranged quadratic data array is expected in the ALPHA register. After execution of "INV" this data array contains the inverse of the original array. The X register displays the conditional number. "INV" changes the stack as well as the registers R00 to R02. Furthermore, "INV" needs just as many extra registers as the data array given in the ALPHA register occupies. If the data array finds itself in the data memory area of the HP-41 (name \( R_{xxx} \)), then "INV" expects these extra data registers after the data array. Example: So that "INV" can calculate the inverse of a data array with the name \( R010 \) of dimension \( 5 \times 5 \), beforehand there has to be at least \( \text{SIZE} \ 61 \), otherwise the message "NONEXISTENT" appears. If the data array is located in the extended memory, the function EMDIR has to show at least 25 empty registers after arranging of the data array, otherwise the message "NO ROOM" will appear.

Description of the function:

The procedure employed calculates the inverse of matrix \( A \) columnwise by solving the equation systems \( A \cdot y_k = e_k \); \( e_k \) are the columns of the unit matrix \( E \), the solutions \( y_k \) are the columns of the inverses \( A^{-1} \).
For the solving of the equation systems an algorithm after Gauss-Jordan is employed. Since the hereby appearing shape changes of matrix "A" have to be used on all "right sides" eₖ, there is produced a matrix whose left half contains the matrix "A" and whose right half contains the unit matrix "E". Since the functions of the CCD-Module store the matrices linewise, the number of lines of the matrix to be inverted is doubled (program lines 4-8). MDIM zeros all newly added elements, so that the matrix will look as depicted below:

```
| a₁₁ | a₁₂ | a₁₃ |
| a₂₁ | a₂₂ | a₂₃ |
| a₃₁ | a₃₂ | a₃₃ |
| 0   | 0   | 0   |
| 0   | 0   | 0   |
| 0   | 0   | 0   |
```

Now all lines are switched in a way so that between two lines of the original matrix there is always an empty line (program lines 9-16):
A sub program, starting at LBL 20, now exchanges the number of lines of the dimension with their column number:

\[
\begin{array}{ccc}
  a_{11} & a_{12} & a_{13} \\
  0 & 0 & 0 \\
  a_{21} & a_{22} & a_{23} \\
  0 & 0 & 0 \\
  a_{31} & a_{32} & a_{33} \\
  0 & 0 & 0 \\
\end{array}
\]

The program lines 18-30 produce the unit matrix in the still empty right half:

\[
\begin{array}{cccc}
  a_{11} & a_{12} & a_{13} & 0 & 0 & 0 \\
  a_{21} & a_{22} & a_{23} & 0 & 0 & 0 \\
  a_{31} & a_{32} & a_{33} & 0 & 0 & 0 \\
\end{array}
\]
The changing of the matrix after Gauss-Jordan happens with two program routines. The outer one, with the pointer $l$ in R01 (initiation in program lines 31-35, starting at LBL 02) now executes the so called pivoting: The function PIV (program line 43) searches the $l$ column from the $l$ to the last line for the greatest absolute element. Now two lines are exchanged in a way that this element gets to the $l$ line (program line 49). Here it is used as a so called pivot in the following inner routine, with the pointer $k$ in R02 (initiation in program lines 37-41), starting at LBL 03. The order R-QR (program line 60) subtracts a manifold of the line of the pivot from the $k$ line in such a way, that the element lying in the column of the pivot becomes equal to 0. The necessary factor for this is calculated beforehand by R-QR by a division of the later to become equal to a 0 element by the pivot (see description of function R-QR). So as to cut down on the errors occuring during the division, the pivoting is executed before. If the number 0 is found as a pivot, matrix "A" was either singular or or least practically singular; because of the occuring round up mistakes the calculator can not distinguish between these. So that during the division the program does not break off with the message "DATA ERROR", a 0 pivot is substituted with a number that is 1E-10 times the greatest absolute in the column of the pivot appearing number, or at least 1E-99 (sub program starting at LBL 30). This number usually lies within the number range of the occuring round up mistakes.

After the working off of both routines, the left half of the matrix, thus the original matrix "A", has been changed into a diagonal matrix:

\[
\begin{array}{cccccc}
    a_{11} & 0 & 0 & z_{11} & z_{12} & z_{13} \\
    0 & a_{22} & 0 & z_{21} & z_{22} & z_{23} \\
    0 & 0 & a_{33} & z_{31} & z_{32} & z_{33} \\
\end{array}
\]
The solution matrix \( A^{-1} \) is reached, when all elements of the right half are divided by the same line positioned diagonal element of the left half. This happens in the routines, starting with LBL 05 and LBL 06, as well as the pointers R 01 on the diagonal element and R 02 on the columns in the right half. If a diagonal element is equal to 0, it is substituted by the greatest absolute element of the column, or at least by \( 1E^{-99} \) (sub program starting with LBL 35). This is possible, because the preceding change made all other elements of the column, except the round up mistakes, equal to 0. Therefore a number is found that is unequal to 0, but still in the number range of the occurring round up mistakes.

After the routines are done calculating the solution \( A^{-1} \) lies in the right half of the matrix. This solution is now transferred to the place of the original matrix \( A \) by exchanging of the dimensions (program line 106), changing of the lines (program lines 107-116) and then finally by redimensioning to a quadratic matrix (program lines 117-120). The conditional number is calculated by multiplication of the frobenius norm of matrix \( A \) before the change (program lines 2-3) with the frobenius norm of its inverses (program lines 121-123).

```
01 *LBL "INV"
02 FNRM
03 STO 00
04 DIM
05 INT
06 LASTX
07 +
08 MDIM
09 1
10 -
11 *LBL 11
12 R<>R
13 1.,001
14 -
15 DSE X
16 GTO 11

17 XEQ 20
18 INT
19 E3
20 /
21 1
22 +
23 IJ=A
24 LASTX
25 *LBL 01
26 R>+
27 RDN
28 >C+
29 ISG Y
30 GTO 01
31 RDN
32 FRC
33 1
```
34 +
35 STO 01
36 ♦LBL 02
37 RCL 01
38 FRC
39 1
40 +
41 STO 02
42 RCL 01
43 PIY
44 ?I J
45 X<>Y
46 X=0?
47 XEQ 30
48 RDN
49 R<>R
50 ♦LBL 03
51 RCL 02
52 INT
53 RCL 01
54 INT
55 X=Y?
56 GTO 04
57 E3
58 /
59 +
60 R-QR
61 ♦LBL 04
62 ISG 02
63 GTO 03
64 ISG 01
65 GTO 02
66 RCL 01
67 FRC
68 1
69 +
70 STO 01
71 ♦LBL 05
72 DIM
73 1
74 +
75 STO 02
76 RCL 01
77 INT
78 1,001
79 *
80 IJ=
81 R>+
82 X=0?
83 XEQ 35
84 RCL 02
85 E3
86 /
87 RCL 01
88 INT
89 +
90 IJ=
91 RDN
92 ♦LBL 06
93 ?I J
94 R>+
95 X<>Y
96 IJ=
97 RDN
98 X<>Y
99 /
100 >R+
101 LAST X
102 ISG 02
103 GTO 06
104 ISG 01
105 GTO 05
106 XEQ 20
107 INT
108 2
109 /
110 0
111 LBL 12
112 1,002
113 +
114 R<>R
115 DSE Y
116 GTO 12
117 INT
118 1,001
119 *
120 MDIM
121 FNRM
122 RCL 00
123 *
124 RTN

125 LBL 20
126 DIM
127 FRC
128 LASTX
129 INT
130 E3
131 /
132 X<>Y
133 LASTX
134 *
135 +
136 MDIM
137 RTN

138 LBL 30
139 RDN
140 RCL 01
141 CMAXAB
142 E10
143 /
144 E-99
145 +
146 RCL Z
147 IJ=
148 X<>Y
149 R+ 
150 RTN

151 LBL 35
152 RCL 01
153 CMAXAB
154 E-99
155 +
156 END

PLNG "INV"
247 BYTES
Chapter 5

Binary Functions
(Hexadecimal Functions)
# Contents Chapter 5

## Hexadecimal Functions

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexadecimal Functions (Introduction)</td>
<td>5.05</td>
</tr>
<tr>
<td>Number Systems</td>
<td>5.05</td>
</tr>
<tr>
<td>The Binary Number System</td>
<td>5.05</td>
</tr>
<tr>
<td>The Octal Number System</td>
<td>5.07</td>
</tr>
<tr>
<td>The Decimal Number System</td>
<td>5.08</td>
</tr>
<tr>
<td>The Hexadecimal Number System</td>
<td>5.09</td>
</tr>
<tr>
<td>Representation of Negative Numbers</td>
<td>5.10</td>
</tr>
<tr>
<td>The Complement</td>
<td>5.10</td>
</tr>
<tr>
<td>Distribution of Ranges of Values</td>
<td>5.11</td>
</tr>
<tr>
<td>Complement (Signed) Modes</td>
<td>5.11</td>
</tr>
<tr>
<td>The 1's Complement Mode</td>
<td>5.12</td>
</tr>
<tr>
<td>The 2's Complement Mode</td>
<td>5.12</td>
</tr>
<tr>
<td>The Unsigned Mode</td>
<td>5.12</td>
</tr>
<tr>
<td>Complement Notation</td>
<td>5.13</td>
</tr>
<tr>
<td>General Conventions for Hexadecimal Functions</td>
<td>5.14</td>
</tr>
<tr>
<td>Basic Setup</td>
<td>5.16</td>
</tr>
<tr>
<td>Functions for Setting Wordsize and Signed Mode</td>
<td>5.17</td>
</tr>
<tr>
<td>WSIZE</td>
<td>5.17</td>
</tr>
<tr>
<td>1CMP</td>
<td>5.19</td>
</tr>
<tr>
<td>2CMP</td>
<td>5.20</td>
</tr>
<tr>
<td>UNS</td>
<td>5.21</td>
</tr>
<tr>
<td>Input and Output Functions for Use with -HEX FNS</td>
<td>5.22</td>
</tr>
<tr>
<td>PMTH</td>
<td>5.22</td>
</tr>
<tr>
<td>VIEWH</td>
<td>5.24</td>
</tr>
<tr>
<td>ARCLH</td>
<td>5.25</td>
</tr>
<tr>
<td>XTOAH</td>
<td>5.26</td>
</tr>
<tr>
<td>Logical Operations</td>
<td>5.27</td>
</tr>
<tr>
<td>AND</td>
<td>5.27</td>
</tr>
<tr>
<td>OR</td>
<td>5.29</td>
</tr>
<tr>
<td>XOR</td>
<td>5.30</td>
</tr>
<tr>
<td>NOT</td>
<td>5.32</td>
</tr>
<tr>
<td>Functions for the Manipulation of Individual Bits</td>
<td>5.33</td>
</tr>
<tr>
<td>S&lt;</td>
<td>5.33</td>
</tr>
<tr>
<td>S&gt;</td>
<td>5.35</td>
</tr>
<tr>
<td>R&lt;</td>
<td>5.37</td>
</tr>
<tr>
<td>R&gt;</td>
<td>5.39</td>
</tr>
<tr>
<td>bS?</td>
<td>5.41</td>
</tr>
<tr>
<td>bC?</td>
<td>5.43</td>
</tr>
<tr>
<td>Cb</td>
<td>5.45</td>
</tr>
<tr>
<td>Sb</td>
<td>5.47</td>
</tr>
</tbody>
</table>
Program Examples ........................................ 5 .49
  Program "W?" (Determination of the Set Wordsize) ...
  Program "CF55" (Clearing of Flag 55) ............... 5 .50
Hexadecimal Functions

The hexadecimal or logical function set consists of the functions 1CMP, 2CMP, AND, bC?, bS?, Cb, NOT, OR, R<, R>, S<, S>, Sb, UNS, WSIZE and XOR. Furthermore these functions are completed by ARCLH, PMTH, VIEWH and XTOAH from the input and output function set (-I/O FNS) of the CCD-Module.

Familiarity with the use of sign modes and number systems in different bases is essential for understanding the descriptions of the functions presented here. Although this knowledge is presupposed, in order to define the meanings and connotations of these concepts to best reflect their use for the functions in the CCD-Module, we shall briefly review them below.

All the hexadecimal functions in the module follow the logical protocol outlined in the flow chart on page 5.15 in their execution, and thus they automatically correct for word size and sign mode.

Number Systems

The Binary Number System

The base of the binary number system is the number 2; all numbers are represented by combinations of the numbers 0 and 1. Binary numbers are denoted in this text by the subscript b; thus 1001_b = 9_d (decimal). The transition to the next higher digits place in binary occurs at 2_d. The table on the next page presents the decimal numbers 0 to 10 and their binary equivalents.
Note that there are two representations given for the binary equivalent of $10_d$. The first is the standard notation that would be obtained, for example by adding $0010_b$ to $1000_b$. The second is the way $10_d$ is represented in the notation referred to as binary coded decimal (or BCD), for which each decimal digit is represented by a separate grouping of 4 binary digits; this is the way the numbers are coded internally by the HP-41.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>10</td>
<td>0001 0000</td>
</tr>
</tbody>
</table>
The Octal Number System

This is a number system of base $8_d$, which utilizes the standard digits 0 through 7 (there is no 8 in this notation); thus the correspondence with decimal 0 through 10 is:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Octal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

Numbers in octal notation in this text will be identified by the subscript $o$. In the octal number system the carryover to the next highest place occurs at decimal 8. Each octal digit represents 3 binary digits (3 bits); through this the octal number system uses the full value range of these 3 places (dec 0 to 7 representing $000_b$ to $111_b$).

The functions OCT and DEC of the HP 41 operating system give the possibility for calculations in the octal number range.
The Decimal Number System

The decimal number system is the customary number system. The functions of the HP-41 support calculations in the base 10 system for all operations. Numbers to base 10 are marked by the subscript d: $136_d$ = decimal number 136. Numbers whose number base has not been clearly given, are assumed to be decimal numbers.
The Hexadecimal Number System

The base of this number system is $16_d$. The numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 as well as the letters A, B, C, D, E and F are used to represent 11 through $15_d$. The carry over to the next highest place takes place at $16_d$. The correspondence between binary, decimal and hexadecimal numbers is:

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>A</td>
<td>1010</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>1011</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>1100</td>
<td>12</td>
</tr>
<tr>
<td>D</td>
<td>1101</td>
<td>13</td>
</tr>
<tr>
<td>E</td>
<td>1110</td>
<td>14</td>
</tr>
<tr>
<td>F</td>
<td>1111</td>
<td>15</td>
</tr>
</tbody>
</table>
Warning:

Hexadecimal numbers "arise" through the use of the full 4 binary digits (4 bits); therefore the hexadecimal number system uses the 4 bits of a nybble in the full value range. Each nybble can represent values from bin 0000 to 1111, in hexadecimal from $0_{h}$ to $F_{h}$. This number system is often used in the computer field.

Representation of Negative Numbers

In a given, limited digit number $m$ (in this case between 0 and 32 binary digits) the value range of the numbers to be represented is $0 \leq n \leq 2^m - 1$; note that no negative numbers are included in this system of notation. To get rid of this deficiency, a complement is introduced.

The Complement

The complement $KOM$ of a number $X$ is defined as:

$$KOM (X) = K - X$$

for which the value of $K$ is fixed by the chosen complement. Since in the binary number system the usual values of $K$ are $2^m$ and $(2^m - 1)$, we usually speak of the "one's Complement" or the "two's complement". In general complements exist in every number system of base $B$ (for example $B=10$ in the decimal system, $B = 16$ in the hexadecimal system, etc.). For any base $B$ there will be a $(B-1)$ and a $B$ Complement, in the decimal system these are the 9's and 10's Complements.
Distribution of Ranges of Values

If there seems no reason for the existence of complement notation, recall that up to this point we have not dealt with negative numbers. We shall include these by adopting an arbitrary distribution of ranges of values in our number systems: in the binary system, for example, we shall define a negative number as one whose most significant (leftmost) bit is set (has a value of 1). By manipulating number distributions and employing complements, the ranges of values are changed as follows:

before complement

\[ 0 \leq X \leq (B^m)-1 \]

after employing the complement

\[
(B \text{ complementation}): \quad -B^{(m-1)} \leq X \leq B^{(m-1)}-1 \\
((B-1) \text{ complementation}): \quad -B^{s(m-1)}-1 \leq X \leq B^{(m-1)}-1
\]

Complement (Signed) Modes

The transformation of decimally represented numbers into binary numbers and vice versa, varies with the signed mode that is used. The three modes are:

- 1’s Complement mode
- 2’s Complement mode
- unsigned mode

When complement notation is used, it becomes simpler for an arithmetic processor to execute subtraction, since this has now been reduced to an addition operation. In addition, it is quite simple for a calculator to construct the complement of a number in a number in binary notation (see below).
The 1's Complement Mode

The 1's complement of a number is created by subtracting this number from the greatest representable number in the chosen word size, i.e. imagining a word size of 5 bits, the 1's Complement of \((- \, a_5)\) is \((11111 - a_5)\). The processor simply inverts all bits of the original number, it executes the logical function "not". Through this arbitrary but definite segmentation of the value range all negative numbers have their highmost bit set which plays the role of the "-" sign. In the 1's Complement mode the number of positive and negative numbers represented are the same, i.e. even zero has two possible representations: 0 and \(-0\), in binary this would mean 00000 and 11111 (still a word size of 5 bits assumed).

The 2's Complement Mode

The 2's Complement of a number is created by subtraction of this number from the greatest representable number and following addition of 1, i.e., supposing a word size of 5 bit the 2's Complement of \((- \, a_5)\) = \((11111 - a_5) + 1\). The 2's Complement of bin (10001) is the number bin (01111). We notice, that by segmenting even in the 2's Complement mode the very left bit is set for all negative numbers and therefore takes over the role of the "-" sign. In the 2's Complement mode there is one more negative number than in in the positive number range, since zero only has the representation 0.

The Unsigned Mode

Since the Complement mode employs a bit as the negative sign, the range of values for a word size of 8 bits in the 1's complement is from \(+127\) to \(-127\) or \(+127\) to \(-128\) for 2's complement. Although these are 256 values sometimes only the positive number range is needed. In this case the Unsigned mode is used, having no preceding sign bit. Therefore, assuming the same wordsize of 8 bits, the value range from dec 0 to 255 is covered.

In the table below, decimal values are assigned to the corresponding binary numbers in the three sign modes for a four bit word size.

(C) Copyright W&W Software Products GmbH 1985 5.12
Complement Notation

<table>
<thead>
<tr>
<th>Binary</th>
<th>1’s Complement mode</th>
<th>2’s Complement mode</th>
<th>Unsigned mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1111</td>
<td>-0</td>
<td>-1</td>
<td>15</td>
</tr>
<tr>
<td>1110</td>
<td>-1</td>
<td>-2</td>
<td>14</td>
</tr>
<tr>
<td>1101</td>
<td>-2</td>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>1100</td>
<td>-3</td>
<td>-4</td>
<td>12</td>
</tr>
<tr>
<td>1011</td>
<td>-4</td>
<td>-5</td>
<td>11</td>
</tr>
<tr>
<td>1010</td>
<td>-5</td>
<td>-6</td>
<td>10</td>
</tr>
<tr>
<td>1001</td>
<td>-6</td>
<td>-7</td>
<td>9</td>
</tr>
<tr>
<td>1000</td>
<td>-7</td>
<td>-8</td>
<td>8</td>
</tr>
</tbody>
</table>
General Conventions for Hexadecimal Functions

We shall now consider the arrangement of the hexadecimal functions. All functions in the CCD-Module observe the following protocol:

1) The current wordsize. It is defined with the function WSIZE.

2) The current sign mode. The decimal representation and the internal hexadecimal representation are linked by the complement, or sign mode employed. Before executing a certain arithmetic function the display always decimally represents the number which is first transformed into binary. The binary result is then retransformed into decimal depending on the current mode.

3) The functions only use the integer portion of the decimal number in X.

The principle operation of the hexadecimal functions is shown in the following flow chart:
Flow Chart for the Functions
AND, OR, XOR, NOT, S<, S>, R< and R>
Keynotes to the Flowchart:

1) It is checked, if all for the operation needed values are representable in the current wordsize and the chosen signs.
2) The numbers, decimally represented in the stack of the HP-41, are changed into binary numbers (dependent on the sign mode).
3) The actual operation is executed.
4) It is checked, if an overflow has taken place during the executed operation (check if the carry has been set). The flag 0 is used as a carry bit.
5) The binary numbers are changed to decimal numbers and placed in the stack.

Basic Setup

Following setups are available after first insertion of the CCD-module, or after a "MEMORY LOST" with the CCD-Module in place:

- Wordsize is set to 8 bits.
- Unsigned mode is set.

Description of the functions

To enable the above mentioned conditions at any time, execute the following steps:

UNS turn on the Unsigned mode
CLX WSIZE set wordsize to 8 bits.
Functions for Setting Wordsize and Signed Mode

WISZE

The function WSIZE sets the wordsize $bb$ for all hexadecimal functions of the CCD-Module. The range of $bb$ is 1 to 32 bits. The wordsize is stored in the CCD-Module's I/O buffer. CLX WSIZE clears the wordsize input in the I/O buffer. If the CCD-Module buffer does not exist a wordsize of 8 bits is assumed. The stack is not changed by this function.

Input

X register: $bb$

Examples

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 WSIZE</td>
<td>Selected wordsize 9 bits, operations possible between $0_d$ to $511_d$.</td>
</tr>
<tr>
<td>33 WSIZE</td>
<td>Error message:&quot;DATA ERROR&quot;. Wordsize was not changed, since values greater than 32 are not allowed.</td>
</tr>
<tr>
<td>16 WSIZE</td>
<td>Selected wordsize is 16 bits.</td>
</tr>
</tbody>
</table>
Further Hints

The number in the X register may be negative as well as contain digits after the decimal point; only the absolute integer value is used. "DATA ERROR" will show when numbers outside the range of 32 to -32 are used, "NONEXISTENT" will be displayed when numbers over 1,000 are used. The selected wordsize has no influence on the contents of the stack or the memory registers. When executing WSIZE, the error message "NO ROOM" may occur, since the wordsize is stored in the I/O buffer of the CCD-Module. In this case new memory space has to be made available before selecting the word size.
1CMP

The function 1CMP puts the HP-41 into the 1’s Complement mode. Other sign modes (2’s Complements or Unsigned modes) are cancelled. The contents of the X register are placed in the LASTX register and are then substituted by the new representation of the hexadecimal number; i.e. the number from the X register is first changed into a binary number in the old mode and after switching to the new mode is changed back into a decimal number.

Input
none

Example

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1CMP</td>
<td>The HP-41 is put into the 1’s Complement mode (only valid for binary functions of the CCD-Module).</td>
</tr>
</tbody>
</table>

Further Hints

If the number in the X register with the new sign mode can not be represented, the error message "DATA ERROR X" will result and the Signed mode will not be changed.

Related Functions

2CMP, UNS

(C) Copyright W&W Software Products GmbH 1985
The function 2CMP puts the HP-41 into the 2's Complement mode. Other sign modes (1's Complement or Unsigned mode) are deleted. The contents of the X register are put into the LASTX register and then replaced by the new representation of the hexadecimal number, i.e. the number in the X register is first changed into a binary number in the old mode and after switching to the new mode is changed back into a decimal number.

Input

none

Example

Key sequence Description

2CMP

The HP-41 is put into 2's Complement mode (only valid for binary functions of the CCD-Module).

Further Hints

If the number in the X register can not be represented in the new mode, the error message "DATA ERROR X" will occur and the mode will not be changed.

Related Functions

1CMP, UNS
UNS

The function **UNS** puts the HP-41 into Unsigned mode. The other signed modes (1's or 2's Complement modes) are cancelled. The contents of the X register are stored in the LASTX register and are then replaced by the new representation of the hexadecimal number, i.e. the number in the X register is first changed into a binary number in the old mode and after switching to the Unsigned mode is changed back into a decimal number.

**Input**

none

**Example**

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNS</td>
<td>The HP-41 is put into the Unsigned mode only valid for binary functions of the CCD-Module</td>
</tr>
</tbody>
</table>

**Further Hints**

If the number in the X register can not be represented in the new mode, the error message "DATA ERROR X" will occur and the mode will not be changed.

**Related Functions**

1CMP, 2CMP
Input and Output Functions for Use with -HEX FNS

These functions are also described in the chapter on -I/O FNS, but for completeness these input and output aides for the CCD-Module hexadecimal functions are also explained in this chapter.

**PMTH**

The function PMTH allows for the input of hexadecimal numbers. It writes the equivalent decimal value into the X register. The stack is rolled up before the value is copied to X.

**Input**

The number of the digits to be put in is dependent on the current wordsize.

**Example**

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNS</td>
<td>switches on the Unsigned mode</td>
</tr>
<tr>
<td>CLX WSIZE</td>
<td>sets the wordsize to 8 bits</td>
</tr>
<tr>
<td>PMTH 8E</td>
<td>Input of the value hex 8E. Immediately after input of the last digit (in this case E). The decimal value will be shown in the X register (in this case 142).</td>
</tr>
</tbody>
</table>

Example program:

"ADDRESS"
PMTH

Using these two steps, the message "ADDRESS" will show, after the input of two numbers the program will be continued.
Further Hints

Text can be shown during the execution of PMTH to specify the data wanted for the prompt. The function rejects input values which are too large. Only the keys 0 to 9 and A to F are active. Terminating the function can be done with the backarrow or ON keys.
VIEWH

The function VIEWH views the hexadecimal equivalent of the number in X.

Input

X register : decimal number in the permitted value range (dependent on the current wordsize and the selected mode).

Example

Key sequence Description
142 VIEWH The number in the X register (142) is changed into its hexadecimal equivalent and this is viewed (in this case 8E).

Further Hints

ALPHA data in X generates the error message "ALPHA DATA". If the decimal number in the X register is not representable in the current wordsize or sign mode, the error message "DATA ERROR X" will be indicated. For numbers whose hex representation has less digits than the current wordsize is set for, leading zeros are placed in the most significant digits.

Related Function

ARCLH, PMTH
ARCLH

ARCLH appends the hexadecimal equivalent of the number in the X register to the contents of the ALPHA register.

Input

X register : Decimal number in the allowed value range (dependent on the current wordsize and the selected complement mode).

Example

Key sequence Description

UNS switches on the Unsigned mode
CLX WSIZE set the wordsize to 8 bits
"ABC" Input of the text "ABC" into the ALPHA register
ARCLH appends two hexadecimal numbers to the ALPHA register; in this case 00, i.e., the ALPHA register now contains ABC00".
55 ARCLH The number hex 37 is appended to the ALPHA register

Further Hints

For ALPHA data in X, the error message "ALPHA DATA" will show. If the decimal number in the X register is not representable in the current wordsize and the sign mode, the error message "DATA ERROR X" will be shown.

Related Functions

VIEWH, PMTH
XTOAH

The function XTOAH appends one or more characters with the value of the X register to the contents of the ALPHA register.

Input

X register : Value of the characters to be appended (depends on the current wordsize and sign mode)

Example

Key sequence Description
CLA 10 WSIZE The ALPHA register is erased and a wordsize of 10 bits is set.
340 XTOAH 340 equals 154\textsubscript{h}. Therefore the following characters are appended to ALPHA, the man character (hex byte 01) and "T" (Byte hex 54).

Further Hints

With ALPHA data in X the error message "ALPHA DATA" is generated. If the decimal number in the X register is not representable in the current wordsize and sign mode, the error message "DATA ERROR X" will be indicated.

Related Function

XTOA (function in the Extended Functions module)
Logical Operations

AND

The function AND combines X and Y using the logical AND function, i.e. in the result all bits that were set in both binary numbers at the same time are set, all other bits are cleared (set to zero). The stack is pushed down and the old X value is placed in LASTX (Binarily this would mean for example: 1011 and 0111 is equal to 0011.)

Input

Y register : Operand 1
X register : Operand 2

Example

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Display</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 WSIZE</td>
<td>4.0000</td>
<td>selecting wordsize 4 bits</td>
</tr>
<tr>
<td>3 ENTER 1 AND</td>
<td>1.0000</td>
<td>0011 and 0001 equals 0001</td>
</tr>
<tr>
<td>7 ENTER 8 AND</td>
<td>0.0000</td>
<td>0111 and 1000 equals 0000</td>
</tr>
<tr>
<td>7 ENTER 15 AND</td>
<td>7.0000</td>
<td>0111 and 1111 equals 0111</td>
</tr>
</tbody>
</table>

Further Hints

Using ALPHA data, the error message "ALPHA DATA" be encountered. If the decimal number in the X or Y register is not representable in the current wordsize and sign mode, the error message "DATA ERROR X" or "DATA ERROR Y" will be indicated.
Related Functions

OR, XOR, NOT
OR

The function OR combines X and Y with the logical OR function, i.e., all bits that were set in both individual binary numbers before the execution of this function, are set in the result. The stack is pushed down and the old X value is put down in the LASTX register. (Binarily this would mean for example: 1011 OR 0111 equals 1111.)

Input

Y register : Operand 1
X register : Operand 2

Examples

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Display</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 WSIZE</td>
<td>4.0000</td>
<td>sets wordsize of 4 bits.</td>
</tr>
<tr>
<td>3 ENTER 1 OR</td>
<td>3.0000</td>
<td>0011 or 0001 equals 0011</td>
</tr>
<tr>
<td>7 ENTER 8 OR</td>
<td>15.0000</td>
<td>0111 or 1000 equals 1111</td>
</tr>
<tr>
<td>7 ENTER 15 OR</td>
<td>15.0000</td>
<td>0111 or 1111 equals 1111</td>
</tr>
</tbody>
</table>

Further Hints

Using ALPHA data the error message "ALPHA DATA" will be generated. If the decimal number in the X or Y register is not representable in the current wordsize or sign mode, the error message "DATA ERROR X" or "DATA ERROR Y" will be shown.

Related Functions

AND, XOR, NOT
The function **XOR** connects X and Y with the logical **EXCLUSIVE OR** function, i.e., all bits that were set in only one of the two original binary numbers are set in the result. If a bit is set if both of the operands or, is clear in both of the operands then it will be cleared in the final result. The stack is lowered and the old X value is placed in LASTX. (Binarily this would mean for example: 1011 exclusive or 0111 equals 1100.)

### Input

Y register : Operand 1  
X register : Operand 2

### Examples

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Display</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 WSIZE</td>
<td>4.0000</td>
<td>sets the wordsize 4 bits</td>
</tr>
<tr>
<td>3 ENTER 1 XOR</td>
<td>2.0000</td>
<td>0011 exclusive or 0001 equals 0010</td>
</tr>
<tr>
<td>7 ENTER 8 XOR</td>
<td>15.0000</td>
<td>0111 exclusive or 1000 equals 1111</td>
</tr>
<tr>
<td>7 ENTER 15 XOR</td>
<td>8.0000</td>
<td>0111 exclusive or 1111 equals 1000</td>
</tr>
</tbody>
</table>

### Further Hints

Using ALPHA data, the error message "ALPHA DATA" is generated. If the decimal number in the X or Y register is not representable in the current wordsize or sign mode, the error message "DATA ERROR X", or "DATA ERROR Y" is displayed.
Related Functions

AND, OR, NOT
**NOT**

The function **NOT** inverts all bits of the number in the X register. The old X value is placed in the LASTX register. (binarily this would mean for example: NOT 1011 equals 0100.)

**Input**

**X register**: Decimal number

**Examples**

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Display</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 WSIZE</td>
<td>4.0000</td>
<td>sets wordsize 4 bits</td>
</tr>
<tr>
<td>3 NOT</td>
<td>12.0000</td>
<td>not 0011 equals 1100</td>
</tr>
<tr>
<td>7 NOT</td>
<td>8.0000</td>
<td>not 0111 equals 1000</td>
</tr>
<tr>
<td>15 NOT</td>
<td>0.0000</td>
<td>not 1111 equals 0000</td>
</tr>
</tbody>
</table>

**Further Hints**

Using ALPHA data, the error message "ALPHA DATA" will display. If the decimal number in the X register is not representable in the current wordsize and sign mode, the error message "DATA ERROR X" is displayed. In the 1's Complement mode executing the function **NOT** would carry with it a change of sign (+ to - or - to +).

**Related Functions**

AND, OR, NOT
Functions for the Manipulation of Individual Bits

S<

The function S< shifts the bits of the X register by one bit (one binary place) to the left. The bit that was pushed out is stored in the carry bit (Flag 0). If, before shifting, the very left bit was set, Flag 0 is set, if it was cleared, Flag 0 will be cleared as well. While shifting the bits the value 0 is always pushed in from the right.

Input

X register : decimal number which is binarily representable in the selected wordsize and the set sign mode.

Example

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Display</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 WSIZE</td>
<td>4.0000</td>
<td>setting the wordsize on 4 bits</td>
</tr>
<tr>
<td>UNS 1</td>
<td>1.0000</td>
<td>selecting of the Unsigned mode</td>
</tr>
<tr>
<td>S&lt;</td>
<td>2.0000</td>
<td>0001 becomes 0010, Flag 0 is cleared</td>
</tr>
<tr>
<td>S&lt;</td>
<td>4.0000</td>
<td>0010 becomes 0100, Flag 0 is cleared</td>
</tr>
<tr>
<td>S&lt;</td>
<td>8.0000</td>
<td>0100 becomes 1000, Flag 0 is cleared and stays cleared</td>
</tr>
<tr>
<td>S&lt;</td>
<td>0.0000</td>
<td>1000 becomes 0000, Flag 0 is set</td>
</tr>
<tr>
<td>S&lt;</td>
<td>0.0000</td>
<td>0000 stays 0000, Flag 0 is clear</td>
</tr>
</tbody>
</table>
Further Hints

$S<$ is equivalent to, in the Unsigned mode and the 2's Complement a multiplication by 2. In the 1's Complement the shift to the left in a positive number range also equals a multiplication by 2, whereas in a negative number range it would be equal to a multiplication by 2 and then a subtraction of 1. Using ALPHA data, the error message "ALPHA DATA" will be displayed. If the decimal number in the X register is not representable in the current wordsize and sign mode the error message "DATA ERROR X" is generated.

Related Functions

$S>$, $R<$, $R>$
The function \( S > \) shifts the binary equivalent of the X register by one bit (one binary place) to the right. The pushed out bit is stored in the carry bit, i.e. in Flag 0. If, before moving, the very right bit was set, Flag 0 is set, whereas if it was cleared, Flag 0 will be cleared as well. When shifting the bits the value 0 is always pushed in from the left.

**Input**

X register: Decimal number that is binarily representable in the selected wordsize and the set sign mode.

**Example**

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Display</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 WSIZE</td>
<td>4.0000</td>
<td>set wordsize on 4 bits</td>
</tr>
<tr>
<td>UNS 8</td>
<td>8.0000</td>
<td>selection of unsigned mode</td>
</tr>
<tr>
<td>S&gt;</td>
<td>4.0000</td>
<td>1000 becomes 0100, Flag 0 is cleared</td>
</tr>
<tr>
<td>S&gt;</td>
<td>2.0000</td>
<td>0100 becomes 0010, Flag 0 is cleared</td>
</tr>
<tr>
<td>S&gt;</td>
<td>1.0000</td>
<td>0010 becomes 0001, Flag 0 is cleared</td>
</tr>
<tr>
<td>S&gt;</td>
<td>0.0000</td>
<td>0001 becomes 0000, Flag 0 is set</td>
</tr>
<tr>
<td>S&gt;</td>
<td>0.0000</td>
<td>0000 stays 0000, Flag 0 is cleared</td>
</tr>
</tbody>
</table>
Further Hints

For ALPHA data, the error message "ALPHA DATA" shows. If the
decimal number in the X register is not representable in the
current wordsize and sign mode, the error message "DATA ERROR X"
will be shown.

Related Functions

S<, R<, R>
The function $R<$ rotates on a binary representation of the X register by one bit (one binary place) to the left. The pushed out bit is stored in the carry bit, in Flag 0 and then pushed in again from the right. If, before rotating, the very left bit was set, Flag 0 is set, if it was cleared, Flag 0 will be cleared as well.

Input

$X$ register: decimal number that is representable in the selected wordsize and set sign mode.

Example

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Display</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 WSIZE</td>
<td>4.0000</td>
<td>set wordsize on 4 bits</td>
</tr>
<tr>
<td>UNS 1</td>
<td>1.0000</td>
<td>selecting Unsigned mode</td>
</tr>
<tr>
<td>$R&lt;$</td>
<td>2.0000</td>
<td>0001 becomes 0010, Flag 0 is cleared</td>
</tr>
<tr>
<td>$R&lt;$</td>
<td>4.0000</td>
<td>0010 becomes 0100, Flag 0 is cleared</td>
</tr>
<tr>
<td>$R&lt;$</td>
<td>8.0000</td>
<td>0100 becomes 1000, Flag 0 is cleared</td>
</tr>
<tr>
<td>$R&lt;$</td>
<td>1.0000</td>
<td>1000 becomes 0001, Flag 0 is set</td>
</tr>
<tr>
<td>$R&lt;$</td>
<td>2.0000</td>
<td>0001 becomes 0010, Flag 0 is cleared</td>
</tr>
</tbody>
</table>
Further Hints

With ALPHA data in X, the error message "ALPHA DATA" is displayed. If the decimal number in the X register is not representable in the current wordsize and sign mode, the error message "DATA ERROR X" will be generated.

Related Functions

S<, S>, R>
The function $\text{R}>$ rotates the binary equivalent of the X register by one bit (one binary place) to the right. The pushed out bit is stored in the carry bit (Flag 0) and then pushed in again from the left. If, before rotating, the very right bit was set, flag 0 is set, if it was zeroed flag 0 will be cleared.

**Input**

X register : decimal number that is binarily representable in the selected wordsize and set sign mode.

**Example**

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Display</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 WSIZE</td>
<td>4.0000</td>
<td>set wordsize 4 bits</td>
</tr>
<tr>
<td>UNS 8</td>
<td>8.0000</td>
<td>select Unsigned mode</td>
</tr>
<tr>
<td>$\text{R}&gt;$</td>
<td>4.0000</td>
<td>1000 becomes 0100, Flag 0 is cleared</td>
</tr>
<tr>
<td>$\text{R}&gt;$</td>
<td>2.0000</td>
<td>0100 becomes 0010, Flag 0 is cleared</td>
</tr>
<tr>
<td>$\text{R}&gt;$</td>
<td>1.0000</td>
<td>0010 becomes 0001, Flag 0 is cleared</td>
</tr>
<tr>
<td>$\text{R}&gt;$</td>
<td>8.0000</td>
<td>0001 becomes 1000, Flag 0 is set</td>
</tr>
<tr>
<td>$\text{R}&gt;$</td>
<td>4.0000</td>
<td>1000 becomes 0100, Flag 0 is cleared</td>
</tr>
</tbody>
</table>
Further Hints

For ALPHA data, the error message "ALPHA DATA" is displayed. If the decimal number in the X register is not representable in the current wordsize and sign mode, the error message "DATA ERROR X" will be shown.

Related Functions

S<, S>, R<
bS?

The function bS? makes it possible to check if binary digits in the X register are set or clear. The decimal number is changed to its binary representation and then the specified bit is checked. This questioning is similar to the checking of a single flag, after the number in X was transferred to flags 0-7 with X<>F (X<>F is contained in the extended functions module, or the HP-41CX). This roundabout way through the flags is not necessary in the bit manipulation functions of the CCD-Module. The answer to bS? is "YES", if bit \( bb \) as specified in the X register is set, it is "NO", if the bit is cleared. A stack drop follows, the old X value is written into the LASTX register.

Input

X register : \( bb \) (decimal number; \( 0 \leq bb \leq \text{current wordsize} - 1 \))
Y register : decimal number that is binarily representable in the selected wordsize and the set sign mode.

Examples

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Display</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 WSIZE UNS</td>
<td>4.0000</td>
<td>setting of wordsize 4 bits and Unsigned mode</td>
</tr>
<tr>
<td>7 ENTER 0 bS?</td>
<td>YES</td>
<td>Bit 0 of the binary number 0111 or 7 is set.</td>
</tr>
<tr>
<td>CLX</td>
<td>7.0000</td>
<td>erasing of indication YES</td>
</tr>
<tr>
<td>7 ENTER 3 bS?</td>
<td>NO</td>
<td>The third bit of the binary number 0111 or 7 is not set.</td>
</tr>
</tbody>
</table>
Further Hints

With ALPHA data in X, the error message "ALPHA DATA" displays. If the decimal number in the Y register is not representable in the current wordsize and the sign mode, the error message "DATA ERROR Y" is generated. The error message "DATA ERROR X" comes up if the given bit $bb$ lies beyond the allowed value range. The numbering of the bits takes place from right to left, the rightmost bit is 0 bit. The highest bit number that can be given is always one smaller than the current wordsize (using wordsize 8 bits 0-7 can be given, thus all 8 bits of the binary number). During a program run the next step is executed, if the given bit is set, if it is cleared the next step is skipped.

Related Functions

$bC?$, Sb, Cb
bC?

The function bC? allows us to check if a bit of the binary representation in the Y register is cleared. The number of the bit to be checked is in X. The answer is "YES" if the corresponding bit in the X register, bit bb, is cleared, it is "NO" if the bit is set. A stack drop follows, the old X value is written into the LASTX register.

Input

X register : bb (decimal number; 0 <= bb <= current wordsize - 1)
Y register : decimal number which is binarily representable in the selected wordsize and the set sign mode.

Examples

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Display</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 WSIZE UNS</td>
<td>4.0000</td>
<td>setting of wordsize 4 bits and Unsigned mode.</td>
</tr>
<tr>
<td>7 ENTER 0 bC?</td>
<td>NO</td>
<td>bit 0 of the binary number 0111 b or 7 is not cleared.</td>
</tr>
<tr>
<td>CLX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 ENTER 3 bC?</td>
<td>YES</td>
<td>bit 3 of the binary number 0111 b or 7 is cleared.</td>
</tr>
</tbody>
</table>
Further Hints

For ALPHA data, the error message "ALPHA DATA" is generated. If the decimal number in the Y register is not representable in the current wordsize and sign mode, the error message "DATA ERROR Y" will be shown. The error message "DATA ERROR X" displays, if the given bit \( bb \) lies beyond the allowed value range. The numbering of the bits takes place from right to left, the rightmost bit is the 0 bit. The highest bit number that can be in X is always smaller than the current wordsize by one (using wordsize 8, bit 0-7 can be given, thus all 8 bits of the binary number). In a running program the next program step is executed, if the given bit is cleared, if it is set, the next step is skipped.

Related Functions

bS?, Sb, Cb
**Cb**

The function Cb allows us to clear the a bit of the binary equivalent of the decimal number in Y. The specified bit number is in the X register. A stack drop follows, the old X value is written into the LASTX register.

**Input**

<table>
<thead>
<tr>
<th>X register</th>
<th>bb (decimal number; 0 &lt;= bb &lt;= current wordsize - 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y register</td>
<td>decimal number which is binarily representable in the selected wordsize and the set sign mode.</td>
</tr>
</tbody>
</table>

**Examples**

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Display</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 WSIZE UNS</td>
<td>4.0000</td>
<td>setting of wordsize 4 bits and Unsigned mode</td>
</tr>
<tr>
<td>7 ENTER 0 Cb</td>
<td>6.0000</td>
<td>bit 0 of the binary number bin 0111 was cleared i.e. 0111 became 0110</td>
</tr>
<tr>
<td>7 ENTER 3 Cb</td>
<td>7.0000</td>
<td>bit 3 of the binary number bin 0111 was not changed, since it is already cleared</td>
</tr>
</tbody>
</table>

(C) Copyright W&W Software Products GmbH 1985 5.45
Further Hints

With ALPHA data in X, the error message "ALPHA DATA" will be shown. If the decimal number in the Y register is not representable in the current wordsize and sign mode, the error message "DATA ERROR Y" will be displayed. The error message "DATA ERROR X" is generated if bit \( bb \) lies beyond the allowed value range. The numbering of the bits takes place from right to left, the rightmost bit is always bit 0. The highest bit number that can be given is always one smaller than the current wordsize (using wordsize 8 bits 0-7 can be given, thus all 8 bits of the binary number).

Related Functions

\( bC? \), \( bS? \), \( Sb \)
Sb

The function Sb enables us to set any bit of the binary number representation of the decimal number in Y. The number in the X register in the given bit \( bb \) of the number in the Y register that is to be set. A stack drop follows, the old X value is written into the LASTX register.

**Input**

X register : \( bb \) (decimal number; \( 0 \leq bb \leq \) current wordsize - 1)

Y register : decimal number which is binarily representable in the selected wordsize and the set sign mode.

**Examples**

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Display</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 WSIZE UNS</td>
<td>4.0000</td>
<td>setting of wordsize 4 bits and Unsigned mode.</td>
</tr>
<tr>
<td>7 ENTER 0 Sb</td>
<td>6.0000</td>
<td>bit 0 of the binary number ( 0111_b ) was not changed, since it was already set.</td>
</tr>
<tr>
<td>7 ENTER 3 Sb</td>
<td>15.0000</td>
<td>bit 3 of the binary number ( 0111_b ) was set, ( 0111 ) became ( 1111 ).</td>
</tr>
</tbody>
</table>
Further Hints

With ALPHA data in X, the error message "ALPHA DATA" will show. If the decimal number in the Y register is not representable in the current wordsize and sign mode, the error message "DATA ERROR Y" displays. The error message "DATA ERROR X" is shown, if the given bit $bb$ lies beyond the allowed value range. The numbering of the bits takes place from right to left, the rightmost bit is always the 0 bit. The highest bit number that can be given is always one smaller than the current wordsize (using a wordsize of 8, bits 0-7 can be specified, thus all 8 bits of the binary number).

Related Functions

bC?, bS?, Cb
Program Examples

Following you will find two helpful programs that will help you to better understand the CCD-Module binary functions.

Determination of the Set Wordsize

Using the following programs the current wordsize can be calculated:

01*LBL "W?"
02 CLX
03 UNS
04 NOT
05 LN1+X
06 4
07 LN
08 /
09 ST+ X
10 END
Clearing of Flag 55

If the printer is plugged in, usually the printer existence flag (flag 55) will be set. In program sections during which no printer is necessary, it is sometimes sensible to clear flag 55, since programs with flag 55 cleared will run faster. The clearing of this flag can be done with the following program:

01*LBL "CF55"
02 14
03 PEEKB
04 0
05 Cb
06 POKEB
07 END

When executing a printer function or a flag 55 question instruction with the printer plugged in, flag 55 will automatically be set or if the program stops running.
Chapter 6

Input/Output Functions
## Contents Chapter 6

### Input/Output functions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>6.05</td>
</tr>
<tr>
<td>Input Functions</td>
<td>6.05</td>
</tr>
<tr>
<td><strong>INPT</strong></td>
<td>6.05</td>
</tr>
<tr>
<td>Programs:</td>
<td></td>
</tr>
<tr>
<td>&quot;INP&quot; (Input to Blocks of Data Registers)</td>
<td>6.05</td>
</tr>
<tr>
<td>&quot;PHINPT&quot; (Input of pH values)</td>
<td>6.07</td>
</tr>
<tr>
<td><strong>PMTH</strong></td>
<td>6.09</td>
</tr>
<tr>
<td><strong>PMTH</strong></td>
<td>6.11</td>
</tr>
<tr>
<td>Program:</td>
<td></td>
</tr>
<tr>
<td>&quot;H-O&quot; (Transformation from HEX to Octal Notation and vice versa)</td>
<td>6.11</td>
</tr>
<tr>
<td><strong>PMTK</strong></td>
<td>6.13</td>
</tr>
<tr>
<td>Programs:</td>
<td></td>
</tr>
<tr>
<td>&quot;KEY&quot; (Menu Control)</td>
<td>6.14</td>
</tr>
<tr>
<td>&quot;?&gt;CAS&quot; (Inquiry)</td>
<td>6.15</td>
</tr>
<tr>
<td>Output Functions</td>
<td>6.17</td>
</tr>
<tr>
<td>Functions Controlling Printer Output</td>
<td>6.17</td>
</tr>
<tr>
<td><strong>ACAXY</strong></td>
<td></td>
</tr>
<tr>
<td>Programs:</td>
<td></td>
</tr>
<tr>
<td>&quot;PR&quot; (Print Example)</td>
<td>6.18</td>
</tr>
<tr>
<td>&quot;TAB&quot; (Expression of Tabulation)</td>
<td>6.19</td>
</tr>
<tr>
<td><strong>PRAXY</strong></td>
<td>6.20</td>
</tr>
<tr>
<td>Programm:</td>
<td></td>
</tr>
<tr>
<td>&quot;PR3&quot; (Formatted Printing)</td>
<td>6.21</td>
</tr>
<tr>
<td><strong>ACLX</strong></td>
<td>6.22</td>
</tr>
<tr>
<td>Programm:</td>
<td></td>
</tr>
<tr>
<td>&quot;PR1&quot; (Print Example)</td>
<td>6.22</td>
</tr>
<tr>
<td><strong>PRL</strong></td>
<td>6.24</td>
</tr>
<tr>
<td>Programm:</td>
<td></td>
</tr>
<tr>
<td>&quot;PR4&quot; (Print Lines)</td>
<td>6.24</td>
</tr>
<tr>
<td>Display of Numbers in Hexadecimal Notation</td>
<td>6.26</td>
</tr>
<tr>
<td><strong>VIEWH</strong></td>
<td>6.26</td>
</tr>
<tr>
<td>Setting the Fix/Eng Display Mode</td>
<td>6.27</td>
</tr>
<tr>
<td><strong>F/E</strong></td>
<td>6.27</td>
</tr>
<tr>
<td><strong>ALPHA Functions</strong></td>
<td>6.29</td>
</tr>
<tr>
<td>Functions for Manipulating Contents of ALPHA</td>
<td>6.29</td>
</tr>
<tr>
<td><strong>ABSP</strong></td>
<td>6.29</td>
</tr>
<tr>
<td><strong>CLA-</strong></td>
<td>6.30</td>
</tr>
<tr>
<td><strong>XTOAH</strong></td>
<td>6.31</td>
</tr>
</tbody>
</table>

(C) Copyright W&W Software Products GmbH 1985 6.03
Output Functions Dealing with the ALPHA Register .............................................. 6 .32
ARCLE .............................................................................................................. 6 .32
Program:
"PR2" (Printout of SI Units) .............................................................................. 6 .32
ARCLH .............................................................................................................. 6 .35
ARCLI .............................................................................................................. 6 .37
In and Output Functions

The functions in this chapter are meant to aid the user in the dialog-oriented programming of the HP-41. These functions give many possibilities for formatted input and output. With this kind of help the user can concentrate fully on the actual solution to the problem, without having to worry about the input and output. In particular, the CCD-Module makes output to HP-IL devices like the Thinkjet printer or Video interface, much easier than ever before. Furthermore the CCD-Module functions help save RAM memory that is needed for other uses, if such functions were written in normal USER-code. Also, the execution time of the programs can be greatly cut down by using the CCD-Module functions.

Input Functions

INPT

The function INPT is a universal data input function. With its help data blocks can be filled very comfortably. This function may be replaced by the following USER-code program:

```
01*LBL 05
02 TONE 0

03*LBL "INP"
04 "I-: "
05 RCL IND 00
06 ARCL X
07 PROMPT
08 CLA-
09 ABSP

A colon and a space are added to the contents of ALPHA
The contents of the register specified by control register R00 are recalled to X and appended to the contents of ALPHA
The contents of ALPHA are displayed and execution is halted; the HP-41 expects numerical input
The original contents of ALPHA are cleared, i.e., from the leftmost
```
10 ABSP character up to and including the colon
11 1
12 X<>Y
13 X<NN?
14 GTO 05
15 2
16 X<>Y
17 X>NN?
18 GTO 05
19 STO IND 00
20 ISG 00
21 END

PLNG "INP"
44 BYTES

As can be clearly seen, using the function INPT saves a lot of complex programming, which would otherwise use up a great amount of program memory. Actually this function consists of two different partial functions. When first executing the function only the first function part is executed. In the above shown USER program this corresponds to the code up to line 07 PROMPT. Then the function INPT executes BST, so that the program pointer points to INPT. In order for the function to know which part of the function to execute, namely the second part, the CCD-Module input flag is set (Bit 5 of byte 4 in Reg. 13).

Now the user is asked to commence with data input, and after pushing the R/S key the second part of the INPT function is executed. This part compares the input value with the input.
boundaries of registers 01 and 02 and stores it, corresponding to the control number in register 00. If no new value was given, the old one is used, but only if it lies within the input boundaries. After this the control number \( iii.fffcc \) in the register 00 is incremented by the value \( cc \) and the next program line is skipped, if: \( iii>fff \) (also see ISG in the HP-41 users handbook).

**Input**

- R000: control number \( iii.fffcc \)
- R001: possible minimum input value
- R002: possible maximum input value

**Examples**

During a chemical experiment 10 different pH values were measured. These are to be put in data registers 10-19 for evaluation by a program that might look like this:

```
01*LBL "PHI
 NPT"
02 10.019
03 STO 00
04 CLX
05 STO 01
06 14
07 STO 02
08 "PH"
09*LBL 01
10 INPT
11 GTO 01
12 BEEP
13 END

PLNG "PHINPT"
 34 BYTES
```

The control number \( iii.fffcc \) is entered and stored in R00
The minimum permitted input (min = 0) is stored in R01
The maximum allowed input (max = 14) is stored in R02
This loop accomplishes input of the 10 data points
Execution is completed
Further Hints

The function **INPT**, like all prompt functions of the CCD-Module, executes **BST** once during the execution in the program. If **BST** is executed at the end of a program, this would take a long time when using large programs. Therefore it is advisable to put all prompt functions of the CCD-Module at the beginning of the concerned program (possibly in a subprogram), or shortly behind a global label. Furthermore the function **INPT**, as well as the function **PMTA**, sets the input flag of the CCD-Module (bit 5 of Reg. byte 13.4). This means that if the function **INPT** was broken off incorrectly (the only right keys are **R/S** and **ON**), the flag is still set, so that when executing the function only the second part is executed. To be sure that the input flag is erased at a certain place in the program, it is best to insert the following program lines before the **INPT** instruction:

```
13.4 These instructions erase the CCD-Module input flag.
    POKEB
```

If only the second part of the function **INPT** is to be executed, the number 1 should be substituted by the number 33.

Related Functions

**ISG**, **PMTA**
**PMTA**

The function **PMTA** gives the possibility of a comfortable ALPHA input. Like the function **INPT** this function consists of two parts. If **PMTA** is executed in a running program, the program run will be interrupted and the program pointer will be set back by one program line to the function **PMTA**. Now the CCD-Module input flag (bit 5 of byte 4 in reg. 13) is set, the ALPHA register is switched and a prompt sign is placed into the display (ALPHA is switched on!). Using **R/S** and **ON** the function can be terminated, without the loss of the original ALPHA contents. If a different key is pushed, all of the previous contents of the ALPHA register are erased, which has no influence on the indication. If the depressed key is a letter key, the ALPHA register will be overwritten with the corresponding letter and this will be appended to the display. After pushing the key **R/S**, **PMTA** is executed for the second time. The function recognizes this by the fact that the input flag of the CCD-module is still set. Now this flag is erased, ALPHA is turned off and flag 23 is set, if there was any input into ALPHA.

**Input**

none

**Example**

The following subprogram clearly shows the effect of the function **PMTA**. In this program the user is asked for his name.
01*LBL "NAME"

02*LBL 01 Beginning of the function loop
03 CF 23 The ALPHA input flag is cleared
04 "NAME: " The HP-41 requests the input of a name
05 PMTA Has the name been supplied?
06 FC? 23 If not, return to LBL 01 and ask again
07 GTO 01 The name has been input and execution is finished
08 END

Further Hints

If, during execution of the function PMTA the ALPHA register is empty, the string "TEXT: " is indicated. Like all prompt functions of the CCD-Module, the function PMTA executes BST once during the execution of the program. If BST is executed at the end of a program, this will take quite a long time when using large programs. Therefore it is advisable to put all prompt functions of the CCD-Module at the beginning of the program (possibly in a sub program), or shortly behind a global label. PMTA also employs the input flag of the CCD-Module (see INPT).

Related Functions

INPT, PMTH, PMTK, PROMPT
The function **PMTH** is the input function of the CCD-Module for hexadecimal numbers. If **PMTH** is executed using the from the keyboard, the user is asked to enter a hexadecimal number, corresponding to the set wordsize. (See *binary functions*) Now the keys 0-9 and A-F are active. If a hexadecimal number is input which is larger than the set wordsize, the function will begin anew and again asks for the input. If the input is correct, the stack is lifted and the number, after having been changed to a decimal number, is written into the X register. If the function is executed in a running program, the ALPHA register is placed in the left of the display and the program run is interrupted. After input of the last hex digit the program execution is automatically continued.

**Input**

The amount of the hexdigits to be given in is dependent on the set wordsize.

**Example**

The following program changes a hexadecimal number into an octal number and vice versa. Asking for the input of the corresponding numbers happens automatically.

```
01*LBL "H-O"     Change from HEX to OKT
02 16            Setting of wordsize to 16 bit. The
03 WSIZE          greatest HEX number therefore is FFFF
04 UNS            which is 177777 octal.
05 "HEX"          Setting of Unsigned mode
06 PMTH           Using "HEX" in ALPHA the function **PMTH**
07 OCT            asks for HEX input.
08 "OCT: "        The decimal number is changed into an
09 ARCLI          octal number and as an integer number is
10 PROMPT         now appended to the ALPHA string "OCT:". Now the octal result is shown.
```

(C) Copyright W&W Software Products GmbH 1985
Further Hints

Like all prompt functions of the CCD-Module, the function PMTH executes BST once during the execution of the program. If BST is executed at the end of a program, it will take quite a long time when using large programs. Therefore it is advisable to put all prompt functions of the CCD-Module at the beginning of the program (possibly in a sub program) or shortly after a global label.

Related Function

ARCLH
PMTK

The function **PMTK** makes it possible to use a menu function for the HP-41. The ALPHA register is displayed and program execution is interrupted. Now the calculator is waiting for the user to press a key. For this there are four different possibilities:

1) The **ON** key turns off the calculator. The program pointer is still set to the function **PMTK**, so that when starting the program at this place the function is executed again.

2) A wrong key is pressed. The calculator answers this with a short sound (only when flag 26 is set).

3) A correct key is pressed. Correct meaning, its ALPHA character is in the display. Additionally, extra text can be placed in the ALPHA register, which has no influence on the menu control. This text must be placed into the ALPHA register, and it is then followed by at least one space and then correct ALPHA character. The function **PMTK** will distinguish between the extra informative text and the correct characters by use of the space, which separates the two text groups from each other. If a key, whose ALPHA character is displayed is pressed, the digit value of the character is written into the X register (the leftmost character being a value of 1). The stack is lifted. This number, being key dependent, can now be used for program ramification. The ALPHA register is erased, except for the commentary text and one empty space.

4) If the ALPHA register is empty, "KEY?" is displayed and the key code (also see ASN and GETKEY from the extended functions module) of the next key that is pressed will be entered into the X register. The stack is lifted.

**Input**

**ALPHA register:** commentary text and correct key signs, separated by at least one empty space or empty ALPHA register.
Example

We want to explain the menu control with a simple example. Controlled by different keys, we want the HP-41 to execute **BEEP** 1-4 times.

01*LBL "KEY"  
02 "1234"  
03 PMTK  
04 GTO IND X  
The correct keys are entered into the ALPHA register. Using **PMTK** "1:2:3:4" is displayed and the calculator is expecting a key to be pressed.  
Depending on the digit value, which in this case corresponds to the ALPHA character we now go to LBL 1-4.

05*LBL 04  
06 BEEP  
07*LBL 02  
08 BEEP  
09*LBL 02  
10 BEEP  
11*LBL 01  
12 BEEP  
13 END
In the following example an application is shown, where the commentary text shall be also be displayed. The user is asked if the given data is supposed to be stored on tape or not.

```
01*LBL "?->C AS"
02 "-TAPE? YN"
03 PMTK
04 X<>Y
05 GTO IND Y
06*LBL 01
07 WRTRX
08*LBL 02
09 END
```

Commentary text and the correct characters (Y and N) are entered into the ALPHA register.

Using PMTK the display will show: "->TAPE? Y:N" with Y for yes and N for NO. The two characters are separated by a colon.

The control number for the function WRTRX is exchanged with the value 1 or 2, which was produced by the function PMTK.

Depending on whether Y or N was pushed, we now go to the corresponding label 01 or 02.

Write to tape.

LBL 02 no operation.

Ready!

If the ALPHA register is empty, the function PMTK works as follows:

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Display</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLA PMTK</td>
<td>KEY?°</td>
<td>asking for a key to be pressed</td>
</tr>
<tr>
<td>SHIFT</td>
<td>31.0000</td>
<td>The key code of the SHIFT key (row 3, column 1 on the keys) is entered into the X register.</td>
</tr>
</tbody>
</table>
Further Hints

If a wrong key is pushed, it can be corrected by pressing it until "NULL" appears in the display. If the ALPHA register is empty, the key code of the key that was pressed, is displayed. If a comma, period or colon are to be one of the characters directly behind the space, at least two empty spaces are needed.

Following extra characters are useful as well:

<table>
<thead>
<tr>
<th>Character code decimal</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>005</td>
<td>R/S</td>
</tr>
<tr>
<td>007, 008</td>
<td>SST</td>
</tr>
<tr>
<td>012</td>
<td>USER, PRGM, ALPHA</td>
</tr>
<tr>
<td>014</td>
<td>SHIFT</td>
</tr>
</tbody>
</table>

Like all prompt functions of the CCD-Module, PMTK executes BST once during the execution of the program. If BST is executed at the end of a program, it may take quite a long while when using large programs. Therefore it is advisable to put all prompt functions of the CCD-module at the beginning of the program (perhaps in a sub program), or shortly behind a global label.

Related Functions

PMTA, PMTH, GETKEY
Output Functions

Functions for Printer Output

ACAXY

With ACAXY (accumulate ALPHA and X by Y) it is possible to transfer text and numbers in the X register to the printer buffer. The contents of ALPHA are justified to the left and the contents of the X register to the right. If flag 20 is set, all of the spaces between the text and the X value are substituted by dots. If text and X value together need more space than the given printer width, the printer width is doubled until there is enough space for an interval. If the ALPHA register is empty, only the X value is transferred (to the right). In FIX/ENG mode the X value is transferred in ENG format with a letter instead of an exponent. After executing the function the X value is stored into the LAST X register and the stack drops. ACAXY works only with the plug in printer, or with an HP-IL module.

Input

X register : Number which is to be transferred in format to the right.
Y register : Printer width (number between 0 and 99)
ALPHA register : Text to be printed to the left.
Examples

Example 1

The following line is to be printed with a 24 character printer.

Width: 200.00 meters

In the program mode the following program steps are necessary:

01*LBL "PR"
02 "WIDTH"
03 18
04 RCL 00
05 ACAXY
06 "METER"
07 ACA
08 PRBUF
09 END

Example 2

The register contents of the registers 1-9 are to be printed in table format using a 24 character printer:

<table>
<thead>
<tr>
<th>Register Contents</th>
<th>Print with TAB Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>R01= 200.00</td>
<td>200.00 15.00 16.20</td>
</tr>
<tr>
<td>R02= 15.00</td>
<td>150.00 20.50 12.20</td>
</tr>
<tr>
<td>R03= 16.20</td>
<td>159.00 18.80 42.32</td>
</tr>
<tr>
<td>R04= 150.00</td>
<td></td>
</tr>
<tr>
<td>R05= 20.50</td>
<td></td>
</tr>
<tr>
<td>R06= 12.20</td>
<td></td>
</tr>
<tr>
<td>R07= 150.00</td>
<td></td>
</tr>
<tr>
<td>R08= 18.80</td>
<td></td>
</tr>
<tr>
<td>R09= 42.32</td>
<td></td>
</tr>
</tbody>
</table>

(C) Copyright W&W Software Products GmbH 1985
Program Listing

01*LBL "TAB"
02 CLA
03 1.009
04 8

05*LBL 01
06 RCL IND
07 ACAXY
08 ISG Y
09 GTO 01
10 PRBUF
11 END

Further Hints

If printer width in the Y register is equal to 0, the standard printer width of 24 characters is assumed. If negative printer widths occur, the absolute value is chosen.

A table of the internationally standardized characters for the exponent is found by looking at the function ARCLE.

Related Function

PRAXY
PRAXY

PRAXY (print ALPHA and X by Y) makes it possible to print text and number in the X register formatted on the carriage width in Y. The contents of ALPHA are printed to the left and the contents of X to the right, depending on the width of the printing field specified in Y. This interval is automatically filled with empty spaces. If flag 20 is set, all spaces between text and X value are substituted by dots. If text and X value together need more space than the given printer width, the printer width is doubled until there is enough space for an interval. If the ALPHA register is empty, only the X value is printed (to the right). In FIX/ENG mode the X value is printed in ENG format with a letter instead of an exponent. After executing the function, the X value is stored into the LASTX register and the stack drops. PRAXY only works with the plug in printer, or with the HP-IL module plugged in.

Input

X register : Value to be printed formatted to the right
Y register : Printer width (number between 0 and 99)
ALPHA register : Text to be printed to the left

Example

The following line is to be printed using a 40 character printer:

NUMBER OF ITEMS......................... 120

(C) Copyright W&W Software Products GmbH 1985 6.20
In the program mode the following program steps are necessary:

```
01*LBL "PR 3"
02 FIX 0
03 "NUMBER OF
   ITEMS"
04 SF 20
05 40
06 120
07 PRAXY
08 END
```

**Further Hints**

If the printer width in the Y register is equal to 0, the standard printer width of 24 signs is assumed. If negative printer widths should occur, the absolute value is selected.

A table with the internationally standardized characters for exponents can be found by looking at the function ARCLE.

**Related Function**

ACAXY
**AACLX**

AACLX (accumulate line by X) transfers *aa* characters with the character code *bbb* into the printer buffer. *aa.bbb* is specified in X. This function makes it possible to output a line of the specified character, this quite fast and saves program memory. AACLX works only with the plug in printer, or with the HP-IL module.

**Input**

X register: *aa.bbb*

**Example**

A number column is to be underlined on the right and the printer stripe is to be limited by two lines, using a 24 character printer:

**Print**

```
***************
  200.00
  -------
***************
```

**Program Listing**

```
01*LBL "PR1"
02  SF 12
03  CLX
04  AACLX
05  RCL 00
06  ACX
07  ADV
08  6.045
09  AACLX
10  ADV
11  .042
12  AACLX
13  ADV
14  END
```
Further Hints

If the value \( aa \) in the X register is equal to 0, the standard printer width of 24 characters is assumed. If negative values should occur, the absolute value is used.

Related Function

PRL
PRL

The function **PRL** (print line) prints a line of 24 (or 12, if flag 12 is set) "-" characters. If any characters were present in the printer buffer, these are printed first, followed by a line feed and then the dashes.

Input
none

Example
The following program clearly shows the effect of **PRL**.

Print

\[
\begin{align*}
\text{TEST1} \\
\text{---------} \\
\text{TEST2} \\
\text{---------}
\end{align*}
\]

Program Listing

\[
\begin{align*}
01*LBL & "PR4" \\
02 & CF 12 \\
03 & "TEST1" \\
04 & ACA \\
05 & PRL \\
06 & SF 12 \\
07 & "TEST2" \\
08 & ACA \\
09 & PRL \\
10 & END
\end{align*}
\]
Further Hints

To reach underlinings of any length the function ACLX must be employed. PRL is only meant for the 24 character printers made by Hewlett-Packard.

Related Function

ACLX
Display of Hexadecimal Number Values

**VIEWH**

The function **VIEWH** (view hex) displays the hexadecimal equivalent of the X value (depends on the set word size and the set mode, see -HEX FNS, otherwise error message "DATA ERROR X" is displayed.)

**Input**

X register : Decimal number in the allowed value range (dependent on the set wordsize and the selected mode).

**Example**

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>142 VIEWH</td>
<td>The number from the X register is changed into its hexadecimal equivalent and displayed (8E).</td>
</tr>
</tbody>
</table>

**Further Hints**

ALPHA data in the X register calls forth the error message "ALPHA DATA", values that are too great cause the error message "DATA ERROR X".

**Related Function**

ARCLH
Setting FIX/ENG Mode

F/E

The function F/E sets the so-called Fix/Eng mode. Flag 40 and 41 are set. The calculator now displays all numbers as in the FIX format. If the number is so large that it has to be expressed with exponents (>9,999,999,999), it is displayed in the ENG format. The number of digits after the decimal point. This F/E mode is looked upon as an indicator for a special printer type by the functions ACAXY and PRAXY (see ACAXY and PRAXY).

Input

none

Example

Number representation in the different modes (for example with two digits after the decimal point):

<table>
<thead>
<tr>
<th>FIX</th>
<th>ENG</th>
<th>F/E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
</tr>
<tr>
<td>1500,00</td>
<td>1,50</td>
<td>1500,00</td>
</tr>
<tr>
<td>89,456,25</td>
<td>89,5</td>
<td>89,456,25</td>
</tr>
<tr>
<td>9,999,999,999</td>
<td>10,0</td>
<td>9,999,999,999</td>
</tr>
<tr>
<td>1,00</td>
<td>100,</td>
<td>100,</td>
</tr>
<tr>
<td>1,59</td>
<td>15,9</td>
<td>15,9</td>
</tr>
<tr>
<td>-7,95</td>
<td>-795,</td>
<td>-795,</td>
</tr>
</tbody>
</table>
Further Hints

The F/E mode stays active as long as no other mode (FIX, ENG or SCI) is selected. If the number of digits after the comma in the F/E mode is to be changed, it is first changed with the function FIX. Afterwards the Fix/Eng mode has to be switched on with F/E.

Related Functions

FIX, SCI, ENG, ACAXY, PRAXY
ALPHA Functions

Functions for Manipulating of the ALPHA Register

ABSP

The function ABSP (ALPHA backspace) erases the rightmost character in the ALPHA register.

Input

none

Example

If the text "ABCDEFG" is in the ALPHA register, the function ABSP erases the right character (in this case "G") and only "ABCDEF" is left in the ALPHA register.

Further Hints

The function ABSP works just as well if the last byte on the right in the ALPHA register has the value 00.

Related Function

CLA-
CLA-

The function CLA- erases the ALPHA register from the right, until it finds a letter followed by an empty space to the right. The space belonging to this letter is not erased.

Input
none

Example
If ALPHA displays for example "KEY 123", after execution of the function CLA- the ALPHA register will display "KEY ". The character string "123" has been erased!

Further Hints
If the ALPHA register is empty or contains only empty spaces or letters, the whole ALPHA register will be erased.

Related Functions
CLA, ABSP
XTOAH

The function **XTOAH** (X to ALPHA hex) appends (depending on the set wordsize) one or more characters to the ALPHA register. These characters correspond to the hexadecimal equivalent of the X register.

**Input**

X register: value of the character(s) to be appended

**Example**

<table>
<thead>
<tr>
<th>Key sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLA 10 WSIZE</td>
<td>The ALPHA register is erased and a wordsize of 10 bits is set.</td>
</tr>
<tr>
<td>340 VIEWH</td>
<td>The value 340 is hexadecimally displayed as '154.</td>
</tr>
<tr>
<td>XTOAH</td>
<td>The characters '01 and '54 (&quot;man&quot; and &quot;T&quot;) are appended to the ALPHA register.</td>
</tr>
</tbody>
</table>

**Further Hints**

Assuming a wordsize of 8 bits the function **XTOAH** works exactly like the function **XTOA** in the Extended Functions module.

**Related Function**

**XTOA** (Extended Functions module)
Functions for Output of the ALPHA Register

ARCLE

The function ARCLE (ALPHA recall engineering) works similar to the function already present in the HP-41 ARCL X (with ENG format), but substitutes the exponent with a letter, corresponding to the internationally standardized SI characters (see table).

Input

X register : Number to be appended to the ALPHA register.

Example

A number with an SI units is supposed to be inserted into the text and then printed on an 80 character printer:

Program

```
01*LBL "PR2" 07 RCL 00 R00= 15,000.00
02 "THE " 08 ARCLE
03 ACA 09 " l-m."
04 "STREET 
  LENGTH " 10 ACA
05 ACA 11 PRBUF
06 "IS " 12 END
```

Print

THE STREET LENGTH IS 15.00 km.

(C) Copyright W&W Software Products GmbH 1985 6.32
# Table of the Internationally Standardized SI Characters

<table>
<thead>
<tr>
<th>Factor</th>
<th>Prefix Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10^{18})</td>
<td>Exa</td>
<td>E</td>
</tr>
<tr>
<td>(10^{15})</td>
<td>Peta</td>
<td>P</td>
</tr>
<tr>
<td>(10^{12})</td>
<td>Tera</td>
<td>T</td>
</tr>
<tr>
<td>(10^{9})</td>
<td>Giga</td>
<td>G</td>
</tr>
<tr>
<td>(10^{6})</td>
<td>Mega</td>
<td>M</td>
</tr>
<tr>
<td>(10^{3})</td>
<td>Kilo</td>
<td>k</td>
</tr>
<tr>
<td>(10^{-3})</td>
<td>Milli</td>
<td>m</td>
</tr>
<tr>
<td>(10^{-6})</td>
<td>Micro</td>
<td>(u ) (instead of (\mu))</td>
</tr>
<tr>
<td>(10^{-9})</td>
<td>Nano</td>
<td>n</td>
</tr>
<tr>
<td>(10^{-12})</td>
<td>Pico</td>
<td>p</td>
</tr>
<tr>
<td>(10^{-15})</td>
<td>Femto</td>
<td>f</td>
</tr>
<tr>
<td>(10^{-18})</td>
<td>Atto</td>
<td>a</td>
</tr>
</tbody>
</table>
Further Hints

If the exponent of the number in the X register is smaller than $10^{-18}$ or greater than $10^{18}$, this number is, as in ARCL X, appended to the ALPHA register with its exponent.

Related Functions

ARCLH, ARCLI
ARCLH

**ARCLH (ALPHA recall hex)** appends the hexadecimal representation of the number in the X register to the contents of the ALPHA register. This representation is dependent on the selected wordsize and the set complement mode (see binary functions). Only characters between 0 and 9 and A to F are appended to the ALPHA register. If the wordsize is unchanged, the number of appended characters is always the same. Depending on the wordsize it may be between 1 and 8 characters.

**Input**

X register: decimal representation of a hex number

**Example**

A hexadecimal number is to be appended to a present text in ALPHA:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLX WSIZE</td>
<td>Setting wordsize on 8 bits</td>
</tr>
<tr>
<td>&quot;VALUE&quot;</td>
<td>The text &quot;VALUE&quot; is written into the ALPHA register.</td>
</tr>
<tr>
<td>PMTH 8E</td>
<td>The hexadecimal number 8E is decimally entered into the X register as value 142.</td>
</tr>
<tr>
<td>ARCLH</td>
<td>The value 142 is appended to the ALPHA register as hexadecimal number 8E. Display shows: &quot;VALUE 8E&quot;.</td>
</tr>
</tbody>
</table>
Further Hints

To avoid mistakes made when using this function, it is advisable to read the chapter binary functions first.

Related Functions

VIEWH, PMTH, XTOAH, ARCLE, ARCLI, ARCL X
ARCLI

The function **ARCLI** (ALPHA recall integer) makes it possible to append only the integer part of a number to the ALPHA register, without changing the display format of the calculator.

**Input**
none

**Example**

ALPHA says for example "COLUMN", the value in the X register is 15.002. After executing the function **ARCLI**, the ALPHA register displays "COLUMN 15".

**Further Hints**
none

**Related Functions**

ARCL X, ARCLE, ARCLH
Chapter 7

Functions for Advanced Programming
### Contents Chapter 7

#### Functions for Advanced Programming

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>7.05</td>
</tr>
<tr>
<td>PLNG</td>
<td>7.05</td>
</tr>
<tr>
<td>PPLNG</td>
<td>7.07</td>
</tr>
<tr>
<td>PHD</td>
<td>7.08</td>
</tr>
<tr>
<td>Functions for Calculating of Absolute Addresses</td>
<td>7.09</td>
</tr>
<tr>
<td>A+</td>
<td>7.09</td>
</tr>
<tr>
<td>A+B</td>
<td>7.11</td>
</tr>
<tr>
<td>A-</td>
<td>7.12</td>
</tr>
<tr>
<td>A-A</td>
<td>7.13</td>
</tr>
<tr>
<td>Decoding Function</td>
<td>7.14</td>
</tr>
<tr>
<td>DCD</td>
<td>7.14</td>
</tr>
<tr>
<td>Program: &quot;CDE&quot; (CODE)</td>
<td>7.15</td>
</tr>
<tr>
<td>Functions for Manipulating the Program Pointer</td>
<td>7.16</td>
</tr>
<tr>
<td>PC&gt;X</td>
<td>7.16</td>
</tr>
<tr>
<td>X&gt;PC</td>
<td>7.18</td>
</tr>
<tr>
<td>X&gt;RTN</td>
<td>7.20</td>
</tr>
<tr>
<td>PC&lt;=&gt;RTN</td>
<td>7.21</td>
</tr>
<tr>
<td>XR&gt;RTN</td>
<td>7.23</td>
</tr>
<tr>
<td>PEEK and POKE Functions</td>
<td>7.24</td>
</tr>
<tr>
<td>PEEKB</td>
<td>7.24</td>
</tr>
<tr>
<td>Programs:</td>
<td></td>
</tr>
<tr>
<td>&quot;A?&quot; (Size of I/O Buffers)</td>
<td>7.25</td>
</tr>
<tr>
<td>&quot;VB&quot; (VIEW BYTES)</td>
<td>7.26</td>
</tr>
<tr>
<td>&quot;GE&quot; (GTO END)</td>
<td>7.27</td>
</tr>
<tr>
<td>PEEKR</td>
<td>7.29</td>
</tr>
<tr>
<td>Program: &quot;VR&quot; (VIEW REGISTER)</td>
<td>7.30</td>
</tr>
<tr>
<td>POKEB</td>
<td>7.31</td>
</tr>
<tr>
<td>Programs:</td>
<td></td>
</tr>
<tr>
<td>&quot;TD&quot; (Tone Duration)</td>
<td>7.31</td>
</tr>
<tr>
<td>&quot;T&quot; (Tone without TIME Module)</td>
<td>7.32</td>
</tr>
<tr>
<td>&quot;TLC&quot;, &quot;TLC1&quot; and &quot;TLC2&quot; (Toggle Lower Case Mode)</td>
<td>7.33</td>
</tr>
<tr>
<td>&quot;CHK&quot; (Correction of Checksum)</td>
<td>7.34</td>
</tr>
<tr>
<td>POKER</td>
<td>7.35</td>
</tr>
<tr>
<td>Program: &quot;CB&quot; (Clear buffer)</td>
<td>7.36</td>
</tr>
<tr>
<td>Sample Programs:</td>
<td></td>
</tr>
<tr>
<td>&quot;ST&quot; (Synthetic Text Lines)</td>
<td>7.38</td>
</tr>
<tr>
<td>&quot;PBC&quot; (Printout of Program Bar Codes)</td>
<td>7.40</td>
</tr>
</tbody>
</table>

(C) Copyright W&W Software Products GmbH 1985 7.03
Functions for Advanced Programming

The 17 functions of this block form the basis for advanced programming techniques, many of which were previously possible only with a great amount of effort, if at all. The following chapter explains each function and demonstrates possible applications example programs. To be able to fully understand this chapter it is important to know about the HP-41 register structure and addressing. These are explained in detail in chapter 1 'Internal Design of the HP-41'.

PLNG

The function PLNG prompts for the input of the program name (global label). In the same way as the function CLP does. Execute the function and press ALPHA name of the label ALPHA. The length of the program in bytes is displayed. No status registers are changed, meaning that the program pointer is unaltered, as are the X register and all the stack registers. As in the function CLP the input sequence PLNG ALPHA ALPHA will display the program length for the current program, which is the one the program pointer is set to.

Input

The program name.
Example

If the program "CDE" is present in the calculator, the key sequence PLNG ALPHA CDE ALPHA will display the message " 60 Bytes". If a printer in TRACE or NORM mode is plugged into the calculator, the number of bytes is printed.

Further Hints

PLNG is not normally programmable although the byte sequence (162, 220) may be entered into a program line. When executed this would normally display the error message "NONEXISTENT". However, if register Q contains a program name backwards the program will stop and display the length of that program in bytes. If register Q is clear then the current program length is displayed, and also program execution is not interrupted.

Related Function

PPLNG
PPLNG

PPLNG (Programmable Program Length) is the programmable version of PLNG. The name of the program is expected in the ALPHA register, or any global label within the program. The result is pushed into the X register, lifting the rest of the stack.

Input

ALPHA register: program name

Example

The program lines
"CDE"
PPLNG

puts the value 60 into the X register, if the program "CDE" is in main memory.

Further Hints

If the ALPHA register is clear, PPLNG shows the length of the current program. When executing PPLNG from the keyboard the program pointer is unchanged. After execution in a program the pointer is moved to the next line to be executed, as would be expected, rather than branching to the program whose length has been calculated.

Related Function

PLNG
PHD

The function PHD (program head) gives the absolute address of the first byte of a program. Like PPLNG, a global label is expected in the ALPHA register. If the ALPHA register is clear the absolute address of the current program is returned. The absolute address of the program head is written into the X register. The format of the address is such that it can be immediately used by PEEKB, POKEB, X>PC or X>RTN.

Input
ALPHA register: program name

Example
Assuming SIZE 000, the key sequence ALPHA name of the first program in CAT 1 ALPHA PHD will return 511.6 in X (on an HP-41 CV or CX).

Further Hints
It should be noted that the absolute address of a program line can change for many reasons and with it the result of PHD. The address changes when the SIZE is changed, often during packing, and all program lines following an inserted program line, including programs further down the catalog chain.

Related Function
PC>X
Functions for Calculation of Absolute Addresses

The calculation functions for absolute addresses are A+, A+B, A-, A-A. These work for the entire memory range of the HP-41, which is from 0.0 to 1023.6. The digits before the decimal point are the absolute register address aaa and the digits after the decimal point are the absolute byte address c within the given register. Values in the X or Y register, that do not correspond to the given range will cause the error message "DATA ERROR X" or "DATA ERROR Y". The digits following the first digit after the decimal point are not used for absolute addresses by the calculation functions, they do not call forth any error messages.

A+

This function increments the absolute address aaa.c specified in the X register by one byte. The result is entered into X, the original value of X before execution of A+ is placed in the LAST X register.

Input

X register: aaa.c

Examples

192.0 A+ results in 192.1
192.6 A+ results in 193.0
Further Hints

The function A+ does not check if the absolute address it calculates actually exists.

Related Functions

A+B, A-, A-A
A+B

This function adds the absolute address $aaa.c$ contained in the Y register to a number ($n$ bytes) specified in X. If the X register contains a negative number, a subtraction will be made corresponding to this. As in a "normal" calculating function, for example $+$, the stack registers are pushed down: The result of the calculation "address plus bytes" is placed in the X register, the amount of the added bytes (original value of the X register) is put into the LAST X register.

Input

X register:  $nnnn$
Y register:  $aaa.c$

Examples

412 ENTER 70 CHS A+B results in 402
192 ENTER 2239 A+B results in 511.6
(= entire main memory range of the HP-41 CV or CX.)

Further Hints

The function A+B works independent of the fact that the absolute address actually exists, for the entire calculating range of 0,0 to 1023,6. The greatest correct input for the X register (amount of the bytes that are to be added) is 7167 (or -7167).

Related Functions

A+, A-, A-A
A-

This function decrements the absolute address *aaa.c* specified in X, by one byte. The result is entered into X, the original value of the X register, before execution of A-, is placed into the LAST X register.

**Input**

X register: *aaa.c*

**Example**

192.1 A- results in 192.0  
193.0 A- results in 192.6

**Further Hints**

The function A- works independent of the fact if the absolute address really exists, for the entire calculating range of 0.0 to 1023.6.

**Related Functions**

A+, A+B, A-A
**A-A**

This function calculates the difference between an absolute address `aaa.c` in the Y register and an absolute address `ddd.e` in the X register. The result in bytes is entered into X. As in a "normal" subtraction, the stack registers are pushed down; the absolute address, which the X register contained before the execution of A-A is entered into LAST X. If, before the execution of A-A, the value in the Y register was greater than the value in the X register, the result will become negative.

**Input**

X register: `ddd.e`
Y register: `aaa.c`

**Example**

511.6 ENTER 192 A-A results in 2239
402 ENTER 412 A-A results in -70

**Further Hints**

The function A-A works independent of the fact if the absolute addresses actually exist, for the entire calculating range of 0.0 to 1023.6.

**Related Functions**

A+, A+B, A-
DCD

The function **DCD** (decode) decodes the value in X and appends the hexadecimally decoded representation to the ALPHA register. This function is especially useful for the analysis of non normalized numbers.

**Input**

X register: value to be decoded

**Example**

**CLA RCL b DCD ALPHA**

Result for example 000000000060F7, this means: The register b did not contain a return address, the program pointer was positioned to 247.5 (according to PC>X).

**Further Hints**

In the result representation of **DCD**, two hexadecimal numerals correspond to one byte, therefore the representation always has 14 digits and every single byte is definitely identifiable.

The return function of **DCD** would be "CDE" (code). The following program codes a hexadecimal representation from the ALPHA register and writes the result into the X and the M register (visible as text):
01 LBL "CDE"
 02 CLX
 03 WSIZE
 04 6,5
 05 SF 22

06 LBL 00
 07 5
 08 PEEKB
 09 ABSP
10 X<>Y
11 RDN
12 57
13 -
14 X<>Y
15 2
16 X<=0?
17 9
18 +

19 X<0?
20 CLX
21 FS?C 22
22 GTO 00
23 16
24 *
25 +
26 POKEB
27 SF 22
28 CLX
29 5.5
30 X<>Y
31 A-
32 X≠Y?
33 GTO 00
34 RCL [ ]
35 END

PLNG "CDE"
60 BYTES

(C) Copyright W&W Software Products GmbH 1985
Functions for Manipulating the Program Pointer

The group of functions that make manipulations of the pointer and the return addresses possible, will be explained next. The functions PC<>RTN, PC>X, X<PC, X>RTN, and XR>RTN allow program manipulation of the program counter and the return stack. This is available for the entire memory range, the status registers and especially to the extended memory of the extended functions module and X memory modules. Additionally, XR>RTN allows a program to jump into any section of code in a plug in software module.

PC>X

The function PC>X (program counter to X) reads the absolute address of the program pointer out of the status register b and writes it into the X register. Since the value is a decimal number it can be stored in a data register. This function makes it possible to calculate any kind of byte distances very easily. It is, for example, possible to recognize if the jumping range of a two byte GTO is long enough for the required jump.

Input
none

Example

We want to calculate the leap length, to find out if a two byte GTO will suffice for this distance. This calculation could be done with the following steps:

Go to the program line after GTO NN, PRGM off, PC>X. Then go to LBL NN; PC>X, A-A.
If the displayed value is smaller than 112, a two-byte-GTO will be able to hold the jump distance when the jump is to be compiled. In case the result was negative (in the program LBL NN is in front of GTO NN), it is necessary before to go to the first PC>X on the program line which is before GTO NN, to exactly calculate the leap distance.

Further Hints

none

Related Functions

PHD, X>PC
X>PC

This function (X to program counter) sets the pointer to the absolute address \textit{aaa.c} given in the X register. Every existent absolute address may be given, not only in the "normal" program memory range. \textit{X>PC} is meant especially for reaching a certain place in a program where the absolute address of the program pointer was beforehand determined by \textit{PC>X}. If this method is employed in the program memory range, the amount of the data registers may not be changed in the meantime, since otherwise the absolute addresses of the program would change.

Input

X register: \textit{aaa.c}

Example

01 PC>X
02 TONE
03 X>PC

This is a neverending program routine and corresponds to the program sequence \textit{RCL b, TONE N, STO b}, except the address returned by \textit{PC>X} can be, contrary to the value from \textit{RCL b}, stored in a data register, and the value for \textit{X>PC} can be a simple decimal number without causing any problems, whereas the value for \textit{STO b} must be encoded before it may be stored away.

Further Hints

\textit{X>PC} is especially useful for directly reaching all of the extended memory of the extended functions or X memory module. The absolute address of the first byte in the extended functions module is 191.6. Here the header of the first file starts. Each file has two header registers and then starts with its actual contents. The absolute addresses are, like in the program memory, descending.
Example

CAT '4 displays:
KA  K004 (Keyfile 4 register)
CLK K000 (Keyfile 0 register)
ALM B006 (Bufferfile 6 registers)
K?  P007 (Program file 7 registers)

99.0000 (free registers in the extended functions module)

(The explanations to the file types K and B are given in the function block X/F memory functions; the program for the production of the file "CLK" with 0 registers can be found there as well.)

Now we want to directly execute the program "K?" in the file "K?" with the function X>PC. For this the absolute address of the program start in the program file must be calculated:

First byte in the extended functions module : 191.6
minus two header registers of "KA" results in : 189.6
minus four register files "KA" results in : 185.6
minus two header registers of "CLK" results in : 183.6
minus zero registers of "CLK" results in : 183.6
minus two header registers of "ALM" results in : 181.6
minus six registers of "ALM" results in : 175.6
minus two header registers of "K?" results in : 173.6

This is the absolute address of the first byte of the program "K?" as a program file in the extended functions module. With 173.6 in the X register and X>PC in this case, we get directly into the program "K?" and can execute it immediately. If the program is not to be executed starting from the first program line, the absolute address of each program line can be determined (for example go through program with SST and then read the program pointer with PC>X).

Related Functions

PC>X, PHD
**X>RTN**

The function **X>RTN** (X to return stack) sets the first return address to the absolute address `aaa.c` specified in X.

**Input**

X register: `aaa.c`

**Further Hints**

Like in **X>PC**, this function can display all actually existing absolute addresses (not only the ones in the program memory). At the next **RTN** or **END** the pointer is set to the given address. Each one of the used return addresses can be stored; this enables us to construct a practically unlimited return stack. Since ramification only occurs at **RTN** or **END**, all stack registers may be set up as necessary.

**Related Functions**

**PC<>RTN**, **XR>RTN**
PC<>RTN

This function (program counter exchange return address) exchanges the present value of the pointer with the present value of the first return address. The function PC<>RTN may therefore only be executed, if there is at least one return address present (otherwise "DATA ERROR").

Input

none

Example

The function sequence

Program A
rrr.b
X>RTN
PC<>RTN
Program B

has the following function:

First program A is executed, then the absolute address rrr.b is stored as the first return address, and PC<>RTN exchanges the pointer with this first return address. Thus the program execution is continued from the absolute address rrr.b (for example in the Extended Functions module!), until RTN or END is reached. Now the program B is executed, since PC<>RTN stored the program pointer at the start of program B as the first return address.
Further Hints

The function PC<>RTN makes it possible to program a subroutine call for any absolute address. A subroutine call and a return from the Extended Functions module are also made possible.

Related Functions

X>RTN, XR>RTN
**XR>RTN**

This function allows the subroutine call or a jump into the program with an XROM number *kk.ll* of a software module at any program line *nnn* (not only where there is a label!). Therefore this function makes it possible to evade the PROMPT functions in some modules or to use only parts of programs as subroutines.

**Input**

- **X register:** program line number *nnn*
- **Y register:** XROM number *kk.ll*

**Example**

The program "T1" in the PPC ROM starts with program line 140, contains first 13 synthetic tones, then an RTN. It has the XROM number 10,47. Usually only XROM "T1" is able to execute all 13 tones together. The function XR>RTN now allows us to execute only part of program "T1". The sequence 10.47 ENTER 150 XR>RTN for example, only executes four tones (lines 150–153).

**Further Hints**

If a return into the main program is desired, PC<>RTN needs to be used after XR>RTN.

**Related Functions**

PC<>RTN, X>RTN
PEEK and POKE Functions

The functions for the pointer manipulation offer a variety of new programming techniques which could only be hinted at in the previous paragraph. For example it is even possible to run programs below the .END., these programs are not listed in CAT‘1 and usually not immediately discovered. To be able to program there, the functions PEEKB, POKEB, PEEKR and POKER, which are talked about in this section, are needed. These functions allow every conceivable byte manipulation in the entire memory range of the calculator. This opens up more and more possibilities as will be shown in several programs.

PEEKB

This function reads the decimal value of the byte, whose absolute address $aaa.c$ is displayed in the X register. The result is placed in X and the stack is lifted. The LAST X register is not changed. The absolute address of a byte is to be entered in the format $aaa.c$, in the course of which $aaa$ is the absolute register address and $c$ is the byte position within the register $aaa$.

Input

X register: $aaa.c$
Examples

If there are any User key assignments (from CAT 2 or CAT 3) the byte 192,6 has the value 240. Using PEEKB it can be determined how many key assignment registers are occupied, since byte 6 of any key assignment register is 240. Therefore first 192,6 has to be tested, then 193,6 etc., until a value different from 240 results.

Program Example "A?"

Program "A?" determines how many registers all in all are occupied with key assignments and I/O buffers (alarm registers, buffers of the CCD Module or other buffers). The correct result can only be obtained if there is at least one free register after the .END.. If there is no unoccupied register, programming will not be easy. If only the amount of the occupied key assignment registers is to be calculated, the program lines from 15-21 need to be deleted.

```
01 LBL "A?"
02 191.6
03 RT
04 240

05 LBL 00
06 RT
07 RT
08 ISG X
09 PEEKB
10 RT
11 X=Y?
12 GTO 00
13 RT
14 RT
15 -.1

16 LBL 01
17 +
18 PEEKB
19 X≠0?
20 GTO 01
21 CLX
22 192
23 -
24 INT
25 END
```

PLNG "A?"

47 BYTES
Another Program Example for PEEKB:

"VB" (VIEW BYTES)

The program "VB" makes it possible to clearly identify text lines in programs or to decode status registers in decimal or hexadecimal form. If the output is desired to be in hexadecimal format, line 11 is to be substituted by ARCLH, and CLX, WSIZE should be executed before starting the program to set the wordsize to 8 bits.

The program "VB" displays any amount of bytes in the ALPHA register in the following format:

absolute byte address - - - decimal value of the byte

The program can be used to display any byte in the memory range of the calculator (except the ALPHA register, since the program changes the ALPHA as well as the stack registers).

Input would be:

Number of desired bytes, ENTER, absolute address of the first byte. If a program is to be completely decoded, the following function sequence is possible:

Enter program name in ALPHA, then PPLNG, PHD, XEQ"VB".

```
01 LBL "VB"
02 LBL 00
03 " "
04 RCL d
05 FIX 1
06 ARCL Y
07 STO d
08 RDN
09 "- - -"
10 PEEKB
11 ARCLI
12 AVIEW
13 RDN
14 A-
15 DSE Y
16 GTO 00
17 CLST
18 CLD
19 END
```

PLNG "VB"
40 BYTES
The output of the program "VB" itself is, for example, is as follows (in the course of which the given absolute addresses are displayed differently, depending on where in memory "VB" resides):

\[
\begin{align*}
25 & @a.1 \_ 1935 \\
27 & @b.8 \_ \_ \\
27 & @c.3 \_ 243 \\
27 & @d.5 \_ \_ \\
27 & @e.4 \_ 86 \\
\end{align*}
\]

One more useful program using PEEKB: "GE"

This program sets the pointer on the first line of the last program in main memory, thus to the program which contains .END. as its last program line. This makes its function identical to its namesake program in the PPC ROM, only much shorter and without synthetics. Therefore the last program is reached quickly (even if it does not have a global label!), without having to run through the entire CAT 1.

In lines 02 to 13 the absolute address of the .END. is decoded out of the bytes 13.0 and 13.1 (register c). Two A+’s result in the absolute address of the start of .END.; this address becomes the first return address and after CLST END executes the jump to the .END., which stops the program run. Now if we switch to the PRGM mode, line 01 of the last program can be seen. (If GTO.. was executed before, this last program contains only the .END.)

```
01 *LBL "GE" 11 *
02 13 12 R↑
03 PEEKB 13 +
04 X<>Y 14 A+
05 A+ 15 A+
06 PEEKB 16 X>RTN
07 16 17 CLST
08 MOD 18 END
09 LASTX
10 X↑2
```

PLNG "GE" 33 BYTES

(C) Copyright W&W Software Products GmbH 1985 7.27
Further Hints

none

Related Functions

PEEK, POKE, POKER
PEEKR

This function can be compared to the RCL function. However, it is now possible to read the contents of any register, without normalization, into the X register. This removes one of the main problems of synthetic programming. The address of the register to be read is entered as absolute address \textit{aaa} into X. As when using an RCL IND X, the stack registers are changed.

Input

X register: \textit{aaa}

Example

Program Example for PEEKR: "VR" (VIEW REGISTER)

The program "VR" shows any number of register contents in hexadecimally decoded format or prints the results. The absolute address is shown as well.

Input would be:
Absolute address of the first desired register as the part before the decimal point, and the last desired register as the number after the decimal point. 192.205 for example displays registers 192 to 205. The program "VR" was principally written for the decoding of the key assignment registers or buffer registers, therefore the registers are treated in ascending order. Line 05 appends three spaces to the ALPHA register. The program "VR" is only possible in this simple format, because PEEKR can not be normalized and simply decoded with DCD. Lines 09 to 11 were inserted so that the program will run faster when the printer is being used (no scrolling).
Further Hints

**PEEKR** works for every existing register address from 0 to 1023, if we want to use data (relative) register addresses with **PEEKR**, the absolute addresses of the data registers must be calculated.

Related Functions

**PEEKB, POKEB, POKER**
POKEB

This function writes over the byte, whose absolute address \( \text{aaa.c} \) is specified in \( Y \), with the value \( \text{bbb} \) specified in \( X \). POKEB works for the entire memory area of the calculator. The stack registers remain unchanged, as long as they are not specified by the absolute address in \( Y \).

Input

X register: \( \text{bbb} \)
Y register: \( \text{aaa.c} \)

Examples

As the first program example we want to show a program for the production of synthetic tones and for the measuring of the tone duration, which changes itself depending on the input value. Thus we are talking about a "self programming" program.

```
01 *LBL "TD"
02  " 
03 ARCLI
04 "F___"
05 PC>XH
06 11
07 CHS
08 A+B
09 X<>Y
10 POKEB
11 TIME
12 TONE X
13 TIME
14 X<>Y
15 HMS-
```

```
16 7 E-6
17 -
18 E4
19 *
20 RCL d
21 FIX 2
22 ARCL Y
23 STO d
24 R↑
25 R↑
26 AVIEW
27 END
```

```
PLNG "TD"
55 BYTES
```
The Program "TD" (Tone Duration)

The program "TD" executes every tone in the range of 0 to 127, by changing the second byte of the function in line 12. If, for example, the program is started with 120 in the X register, line 12 is changed into TONE P.; this change is brought about by the function POKEB, which is in line 10! The rest of the program consists of the time measuring with the help of the function TIME of the TIME module. If the principle of POKEB, as represented here, is to be used without the time measuring with the TIME module, the program is to be shortened accordingly:

```
01 LBL "T"
02 PC > X
03 - 8
04 A + B
05 X <> Y
06 POKEB
07 TONE X
08 END
```

Any TONE function may be entered into the program at line 07. Before executing "TD" or "T", the program memory must be packed, to exclude any possibility of an error.

A program using the functions PEEKB and POKEB is "TLC". It serves the function of switching, enabling and disabling the lower case mode of the CCD Module. If ALPHA mode is on and USER mode is off, it is possible to enter any ALPHA characters. This special function can be deactivated or activated again with the program "TLC". The corresponding coding takes place in register c (13) byte 4. Inputs are not necessary for "TLC".
As can be seen when looking at the program example, there are several different possibilities to switch a bit around.

Another especially useful program using PEEKR, PEEKB and POKEB is "CHK". It corrects the checksum of a program file in the Extended Functions module. If an uncompiled program (not all GTO's and XEQ's were executed before the storing into the extended functions module) is directly executed in the extended functions module, it will not be possible to get the program back into main memory using GETP or GETSUB. If this is attempted it will call forth the error message "CHKSUM ERR". Now only the program "CHK" has to be executed with the name of the program file in the ALPHA register, and the checksum is corrected. The program file must be completely in the Extended Functions module! The program "CHK" may also be used for programs, which are completely inside an X memory module.
Now line 03 should be changed to 751 or 1007, depending in which port the X memory module has been plugged. Line 04 appends six spaces to the ALPHA register.

01 LBL "CHK"
02 RCL PT A
03 191
04 "T"
05 7
06 AROT
07 RCL [ ...
08 LBL 00
09 R†
10 R†
11 PEEKR
12 DSE ¥
13 R†
14 X≠Y?
15 GTO 00
16 CLX
17 STO \n18 RDN

Further Hints

Since POKEB can change any byte, this function should only be employed if the calculation function is clear. Otherwise it may draw forth unwanted changes in programs, data registers, status register etc. or "MEMORY LOST".

Related Functions

PEEKB, PEEKR, POKER

(C) Copyright W&W Software Products GmbH 1985
POKER

This function writes over the absolute register, whose address \textit{aaa} is specified in the Y register, with the contents of the X register. \textbf{POKER} works for the entire existing register range of the calculator. The stack registers remain unchanged, as long as they are not specified by the absolute address in the Y register.

\textbf{Input}

X register: value to be stored
Y register: \textit{aaa}

\textbf{Examples}

\textbf{Program Example for POKER: Program "CB" (Clear Buffer)}

The function of the program "CB" corresponds to the program "CK" from the PPC ROM. It erases the entire I/O buffer area. All registers starting with the register with the absolute address 192 up to the register directly under the \texttt{.END.} are erased. This erases all I/O buffers and all key assignments from \texttt{CAT 2} or \texttt{CAT 3}. In \texttt{CAT6} it is necessary to erase the key assignment bit to each key with the key C (assignments appear as \texttt{ABS}). All key assignments bits can be erased at once using 15 \texttt{ENTER CLX POKER} and 10 \texttt{ENTER CLX POKER}; but this also causes all key assignments of the global labels to become ineffective. The actual purpose of the program "CB" is to erase garbage that has been placed in the I/O buffer area.
The program "CB" calculates the absolute address of the register, in which the .END. is stored. Afterwards this address is decreased by one in line 14. If there are some free registers present in the calculator (RTN, PRGM on: displays 00REG NN), these registers can be hidden by storing a number into the register directly under the .END. Thus "CB" is only run until line 14 (incl.) and then the input: ENTER, POKER follows. Now the HP-41 does not show any free registers. This faulty state can be neutralized by executing program "CB".

Every unwanted register within the free registers can be erased with "CB". The program can also be modified in such a way that a certain number of key assignment registers and buffer registers is preserved; for this the number of the registers to be preserved has to be subtracted from 193 and the result has to be inserted in line 17 of the program "CB". Corresponding to this, the programs "A?" and "CB" can be combined.

```
01 LBL "CB"
02 13
03 PEEKB
04 X<>Y
05 +
06 PEEKB
07 16
08 MOD
09 LASTX
10 X↑2
11 *
12 R↑
13 +
14 DSE X
15 E3

16 /
17 193
18 +
19 DSE X
20 CLST
21 .
22 LBL 00
23 POKER
24 ISG Y
25 GTO 00
26 END

PLNG "CB"
46 BYTES
```
Further Hints

Since POKER can change any register, this function should only be employed if the calculating structure is clear. Otherwise it may result in unwanted changes in programs, data registers, status registers, etc. or "MEMORY LOST".

Related Functions

PEEKB, PEEKR, POKEB
The entire function block of the -ADV FNS offers a great number of new possibilities for advanced programming of the HP-41. A further example for this is the program "ST". It serves to find all synthetic text lines of a program decodes them and then prints them. After the prompt "PRGM?" a global label of the program to be checked has to be entered and started with R/S. SIZE 020 is needed. The program "ST" does not contain any synthetic text lines, but does contain three synthetic three byte GTO functions:

line 031 GTO 13
line 121 GTO 08
line 142 GTO 08

The Program "ST" (Synthetic Text Lines)

01 LBL "ST"
02 CLST
03 WSIZE
04 STO 01
05 "PRGM? "
06 PMTA
07 PRA
08 PHD
09 STO 00
10 CF 22
11 GTO 08
12 LBL 01
13 RCL 00
14 A-
15 STO 00
16 LASTX
17 PEEKB
18 RTN
19 LBL 02
20 XEQ 01

(C) Copyright W&W Software Products GmbH 1985
38 LBL 07
39 FS? 22
40 GTO 09
41 SF 22

42 LBL 08
43 ISG 01
44 CLX
45 CF 21
46 VIEW 01

47 LBL 09
48 XEQ 01
49 STO 04
50 X=0?
51 CF 22
52 X=0?
53 GTO 09
54 16
55 X>Y?
56 GTO 08
57 13
58 +
59 X>Y?
60 GTO 07
61 CF 22
62 2
63 +
64 X>Y?
65 GTO 05
66 113
67 +
68 X>Y?
69 GTO 08
70 48
71 +
72 X>Y?
73 GTO 03
74 14
75 +

76 X=Y?
77 GTO 03
78 1
79 +
80 X=Y?
81 GTO 03
82 X>Y?
83 GTO 04
84 33
85 +
86 X>Y?
87 GTO 02
88 -
89 X=0?
90 SF 21
91 4
92 +
93 E3
94 /
95 5
96 +
97 STO 02
98 1
99 -
100 STO 03
101 FS? 21
102 GTO 11

103 LBL 10
104 XEQ 01
105 127
106 X=Y?
107 GTO 11
108 32
109 -
110 X<Y?
111 SF 21
112 X<>L
113 X>Y?
114 SF 21

(C) Copyright W&W Software Products GmbH 1985 7.39
Barcodes of Programs in Main Memory
with the CCD-Module and ThinkJet Printer

The program "PBC" makes it possible to print bar codes of any program directly out of the RAM of the HP-41. For this the extended functions module, the ThinkJet printer (with IL module) and the CCD-module are required.

Program Use:
1. Find the printer and select with CAT'0 and set back. (Key ENTER and key C in CAT'0)
2. SIZE 019 is necessary.
3. A global label of the program, of which the bar codes are to be printed is entered into ALPHA. If the ALPHA register is empty, the program "PBC" prints its own bar codes.
4. Start the program "PBC".
<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>LBL &quot;PBC</td>
</tr>
<tr>
<td>02</td>
<td>PHD</td>
</tr>
<tr>
<td>03</td>
<td>CLRG</td>
</tr>
<tr>
<td>04</td>
<td>STO 00</td>
</tr>
<tr>
<td>05</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>PPLNG</td>
</tr>
<tr>
<td>07</td>
<td>STO 02</td>
</tr>
<tr>
<td>08</td>
<td>-</td>
</tr>
<tr>
<td>09</td>
<td>A+B</td>
</tr>
<tr>
<td>10</td>
<td>PEEKB</td>
</tr>
<tr>
<td>11</td>
<td>SF 23</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>BS?</td>
</tr>
<tr>
<td>14</td>
<td>CF 23</td>
</tr>
<tr>
<td>15</td>
<td>9.018</td>
</tr>
<tr>
<td>16</td>
<td>STO 03</td>
</tr>
<tr>
<td>17</td>
<td>SF 21</td>
</tr>
<tr>
<td>18</td>
<td>PRA</td>
</tr>
<tr>
<td>19</td>
<td>&quot;#F*r784S</td>
</tr>
<tr>
<td>20</td>
<td>ACA</td>
</tr>
<tr>
<td>21</td>
<td>&quot;ν&quot;</td>
</tr>
<tr>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>23</td>
<td>CRFLAS</td>
</tr>
<tr>
<td>24</td>
<td>LBL 00</td>
</tr>
<tr>
<td>25</td>
<td>RCL 00</td>
</tr>
<tr>
<td>26</td>
<td>A-</td>
</tr>
<tr>
<td>27</td>
<td>X&lt;&gt; 00</td>
</tr>
<tr>
<td>28</td>
<td>PEEKB</td>
</tr>
<tr>
<td>29</td>
<td>STO IND</td>
</tr>
<tr>
<td></td>
<td>03</td>
</tr>
<tr>
<td>30</td>
<td>ST+ 06</td>
</tr>
<tr>
<td>31</td>
<td>DSE 01</td>
</tr>
<tr>
<td>32</td>
<td>GTO 01</td>
</tr>
<tr>
<td>33</td>
<td>143</td>
</tr>
<tr>
<td>34</td>
<td>-</td>
</tr>
<tr>
<td>35</td>
<td>X&lt;=0?</td>
</tr>
<tr>
<td>36</td>
<td>97</td>
</tr>
<tr>
<td>37</td>
<td>64</td>
</tr>
<tr>
<td>38</td>
<td>-</td>
</tr>
<tr>
<td>39</td>
<td>X&lt;=0?</td>
</tr>
<tr>
<td>40</td>
<td>34</td>
</tr>
<tr>
<td>41</td>
<td>32</td>
</tr>
<tr>
<td>42</td>
<td>-</td>
</tr>
<tr>
<td>43</td>
<td>X&lt;=0?</td>
</tr>
<tr>
<td>44</td>
<td>3</td>
</tr>
<tr>
<td>45</td>
<td>STO 01</td>
</tr>
<tr>
<td>46</td>
<td>STO 04</td>
</tr>
<tr>
<td>47</td>
<td>LBL 01</td>
</tr>
<tr>
<td>48</td>
<td>ISG 03</td>
</tr>
<tr>
<td>49</td>
<td>X&lt;0?</td>
</tr>
<tr>
<td>50</td>
<td>GTO 02</td>
</tr>
<tr>
<td>51</td>
<td>DSE 02</td>
</tr>
<tr>
<td>52</td>
<td>GTO 00</td>
</tr>
<tr>
<td>53</td>
<td>LBL 02</td>
</tr>
<tr>
<td>54</td>
<td>RCL 05</td>
</tr>
<tr>
<td>55</td>
<td>16</td>
</tr>
<tr>
<td>56</td>
<td>MOD</td>
</tr>
<tr>
<td>57</td>
<td>LASTX</td>
</tr>
<tr>
<td>58</td>
<td>FC? 23</td>
</tr>
<tr>
<td>59</td>
<td>ST+ X</td>
</tr>
<tr>
<td>60</td>
<td>+</td>
</tr>
<tr>
<td>61</td>
<td>ST+ 06</td>
</tr>
<tr>
<td>62</td>
<td>STO 07</td>
</tr>
<tr>
<td>63</td>
<td>ISG 05</td>
</tr>
<tr>
<td>64</td>
<td>CLX</td>
</tr>
<tr>
<td>65</td>
<td>RCL 04</td>
</tr>
<tr>
<td>66</td>
<td>RCL 01</td>
</tr>
<tr>
<td>67</td>
<td>DSE X</td>
</tr>
<tr>
<td>68</td>
<td>-</td>
</tr>
<tr>
<td>69</td>
<td>RCL 08</td>
</tr>
<tr>
<td>70</td>
<td>+</td>
</tr>
<tr>
<td>71</td>
<td>STO 08</td>
</tr>
<tr>
<td>72</td>
<td>RCL 06</td>
</tr>
<tr>
<td>73</td>
<td>+</td>
</tr>
<tr>
<td>74</td>
<td>255</td>
</tr>
</tbody>
</table>
Program Description:

The program uses two routines from the bar code program of Winfried Maschke, as was published in PRISMA 10/11-1982 and in the PPC Calculator Journal V9N4P45. The bar codes are, corresponding to the status of the program of which they are printed, produced in normal and privately protected format. To be sure that the graphic capacity of the ThinkJet printer can never be exceeded only ten program bytes (+ three control bytes) are printed in one row. The program contains eleven synthetic text lines:
The decimal byte values of the program of which the bar codes are to be printed, are read out of the program memory with **PEEKB** (line 28) and stored in the data registers 9 to 18 (line 29). Register 00 contains the absolute byte address of the byte to be worked on next. The number of program bytes is stored in data register 02; this register is used as a counter (lines 51 and 172). If ten byte values are stored, the values for three control bytes are calculated (lines 54 to 78); these are stored in data registers 06, 07, and 08; registers 05 contains the respective row number.

For each single bit of the bar code row a graphic byte is stored into a text file, starting with line 95. Line 95 contains the two bits for the two start bits; all other bits are formed in the labels 07,08 or 09 for each bit of the read program. Each byte of the program gives us 8 to 12 graphic bytes. To reach the desired height of the bar code rows, the entire graphic information of a row is sent to the printer 26 times (Label 05 and Label 06). The height of these bars can be changed using the control number in line 119. Lines 156 to 159 in connection with the escape sequence from line 139 cause a line feed to the next bar code row. This control number (6) is in line 138 and can be changed as well.
The program prints a row of immediately readable bar codes in first quality in about 3 1/2 minutes. If the graphic bytes were not stored in a text file, a program with the same function, such as "PBC" would be possible without the Extended Functions module, but in this case the printing of a bar code row would take about 35 minutes! After 18 bar code rows have been printed on one page, the program automatically executes a form feed (lines 162 to 166); control number for this in line 163. This way the program automatically prints bar codes for programs of any length using endless paper. If single pages are used, a new paper has to be put in by hand after the paper output and the blue key "TOF" has to be used. (For this paper change the program does not have to be stopped.)

Flags 17, 22 and 23 are used. The program can be stopped at any time; it can also be executed using SST. If the calculator is to be turned off during the program run (for example for a battery change), the state of the flags 17, 22 and 23 has to be asked about beforehand. This state must be reconstructed when starting anew. The stack registers and the ALPHA register, the data registers used by the program (0 to 18) and SIZE may obviously not be changed either, if the program is to be executed successfully. If the program is to be broken off, the printer must be cleared (by CAT'0) and the text file should be erased.

If you own an Extended Functions module of the B revision, you should, after the program ends, define a working file (because of PURFL in line 174). The program contains a Long Form GTO (208,000,000) in line 173.

Program (319 bytes) and text by Gerhard Kruse.
Chapter 8

XF/Memory Functions
Contents Chapter 8

XF/Memory Functions

<table>
<thead>
<tr>
<th>The Extended Functions/Memory Functions Block</th>
<th>8.05</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAVEB</td>
<td>8.05</td>
</tr>
<tr>
<td>GETB</td>
<td>8.07</td>
</tr>
<tr>
<td>Program &quot;BS?&quot; (Buffer Size?)</td>
<td>8.08</td>
</tr>
<tr>
<td>SAVEK</td>
<td>8.09</td>
</tr>
<tr>
<td>GETK</td>
<td>8.10</td>
</tr>
<tr>
<td>Program &quot;CLK&quot; (Establishment of a Key Assignment file with 0 Registers)</td>
<td>8.11</td>
</tr>
<tr>
<td>MRGK</td>
<td>8.12</td>
</tr>
<tr>
<td>Program &quot;PK&quot; (Pack Key Assignment Registers)</td>
<td>8.13</td>
</tr>
<tr>
<td>SORTFL</td>
<td>8.15</td>
</tr>
<tr>
<td>Program &quot;WF&quot; (Write and Read File)</td>
<td>8.17</td>
</tr>
</tbody>
</table>

(C) Copyright W&W Software Products GmbH 8.03
The Extended Functions/Memory Functions block

These functions expand the capability and utility of the Extended Functions module. Therefore they can only be employed in connection with this module or an HP-41CX. Otherwise the error message "NO XF/M" will occur.

SAVEB

Saves the I/O buffer with the ID number \( aa \) in the buffer file specified in alpha, \( aa \) is specified in X. If the ALPHA register is empty, the current file is used. The function SAVEB allows us to construct as many files as wanted (with different names) of the same buffer ID number (for a description of the various buffers ID’s in use see B?).

Input

\[
\begin{align*}
\text{X register} & : \quad aa \\
\text{ALPHA register} & : \quad \text{file name}
\end{align*}
\]

Example

The present alarm data are to be stored as a file with the name "ALM".

Input: \text{ALPHA ALM ALPHA 10 SAVEB}.

These inputs cause the bufferfile "ALM" to be constructed and the data in this buffer is saved. The original buffer is preserved in memory. The alarm data may be erased by using 10 CLB, this will free the registers.
Further Hints

A buffer file with the file type B is displayed in using CAT'4 with the CCD Module in the calculator. The CCD module distinguishes between different buffer files, since it is not visible in CAT'4; this means, a buffer file which contains alarm data for the TIME module can not be written over by a buffer file with a different ID number.

Related Functions

GETB, SAVEK, GETK
GETB

This function puts the buffer data from the buffer file back into the I/O buffer. The file name is in the ALPHA register, if the ALPHA register is empty, the current file is employed. If an I/O buffer which corresponds to the buffer file to be retrieved already exists, this I/O buffer is erased before using the file data to construct a new buffer.

Input

ALPHA register: file name

Example

Program for I/O Buffers: "BS?" (Buffer Size)

The program "BS?" is to be started with the desired buffer ID number in the X register. It then calculates how many registers the corresponding buffer occupies. If the number given in X does not correspond to an existing buffer the function SF 99 in line 04 shall cause the error message "NONEXISTENT". The program works perfectly independent of how many different buffers exist. Also any amount (or none) of key assignment registers may be occupied. The ID number of the buffer which is constructed by the CCD Module itself, is 5. Therefore 5 XEQ "BS?" indicates how many registers this buffer occupies.
Further Hints

If the calculator is switched on without the module that created the I/O buffer, the management system of the HP-41 automatically erases that I/O buffer. But the functions SAVEB and GETB work independent of the fact if the module belonging to the treated buffer is plugged in or not.

The function CLFL should not be executed for buffer files; for erasing the files, PURFL should be used.

Related Functions

SAVEB, SAVEK, GETK, MRGK
SAVEK

This function stores all key assignments of functions from CAT 2 and CAT 3 as a keyfile in extended memory.

The use is analogous to SAVEP. The file name must be specified in the ALPHA register. If the ALPHA register is empty, the current file is employed.

Input

The existing key assignments from CAT 2 and CAT 3 are to be stored in a keyfile with the name "KEYS". For this the following steps are necessary:

ALPHA "KEYS" ALPHA SAVEK

Further Hints

A keyfile is displayed in CAT'4 with the display K. The key assignments are preserved unchanged.

Related Functions

GETK, MRGK, GETB, SAVEB

(C) Copyright W&W Software Products GmbH 8.09
GETK

This function erases all key assignments of functions from CAT 2 and CAT 3, and then activates the stored key assignments in the specified key file. The file name is in the ALPHA register. If the ALPHA register is empty, the current file is used.

Input

ALPHA register: file name

Example

A Program for GETK: "CLK" (Constructing of a CLEAR KEY FILE)

The program "CLK" constructs a file which is displayed as "CLK K000" in CAT'4, it consists only of header registers. This file erases all key assignments of functions from CAT 2 and CAT 3. (CLKEYS would erase all key assignments of global labels as well.) This program does not require any input. It can only be used if there are at least two free registers in the extended functions portion of extended memory. Otherwise the program will stop in line 10 displaying "NONEXISTENT" or at line 17 displaying "DATA ERROR".

The program "CLK" contains two synthetic text lines:

line 03: F7, FF, FF, FF, FF, FF, FF, FF

line 30: F0 (NOP).

line 18 contains the file name "CLK" and four spaces; meaning the ALPHA register has the length 7!

Immediately after "MEMORY LOST" or after the extended function module was plugged in anew, it is necessary to execute CAT'4 once, before the program "CLK" can be used.
A register with seven hex FF bytes (dec 255) is stored as a limit below the last employed register in the extended memory. First, the program "CLK" searches for this border (lines 02 to 13). Now this border register is written over with the file name "CLK" (lines 18 to 20), and the register immediately below gets the criterion as keyfile (lines 22 to 28). The register directly below is now marked as a new border register (lines 29 to 32). Now GETK, with the help of the produced keyfile, erases all key assignments of functions from CAT 2 and CAT 3.

Further Hints

Key assignments of programs are only erased if a different assignment is put on the corresponding key by the fetched keyfile.

Related Functions

SAVEK, MRGK, SAVEB, GETB
MRGK

This function activates the key assignments stored in the employed keyfile. Existing key assignments are only erased if the same key is occupied by a key in the keyfile. Other existing assignments remain unchanged.

Input

ALPHA register: file name

Example

Since the function GETK erases all existing key assignments from CAT 2 and CAT 3, the functions SAVEK and GETK give a simple and quick possibility to pack the key assignment registers.

The key sequence for this is:

Put any name into ALPHA, SAVEK, GETK, PURFL. If you possess an Extended Functions module of the revision B, CAT'4 or EMDIR should be executed following this, to secure the files in extended memory.

The following program packs the key assignment registers only by use of the functions PEEKB and POKEB. The program needs no input and can be stopped at any time or executed with SST. But if the function POKEB has been reached once, the program must run through until the end, since otherwise there will be chaos or double storages in the assignment registers. The program "PK" requires SIZE 002. All buffers remain untouched.
01 LBL "PK"
02 192
03 STO 00
04 STO 01

05 LBL 00
06 RCL 00
07 ,6
08 +
09 PEEKB
10 240
11 X≠Y?
12 GTO 01
13 RCL 00
14 PEEKB
15 X≠0?
16 XEQ 02
17 RCL 00
18 ,3
19 +
20 PEEKB
21 X≠0?
22 XEQ 02
23 ISG 00
24 CLX
25 GTO 00

26 LBL 01
27 RCL 00
28 RCL 01
29 X=Y?
30 GTO 05
31 0
32 POKEB
33 X<>Y
34 A+
35 X<>Y
36 POKEB
37 X<>Y

38 A+
39 XEQ 03
40 GTO 01
41 LBL 02
42 RCL 01
43 XEQ 04
44 XEQ 04
45 LBL 03
46 X<>Y
47 POKEB
48 X<>Y
49 ENTER
50 FRC
51 +
52 ENTER
53 FRC
54 4
55 /
56 -
57 STO 01
58 RTN
59 LBL 04
60 X<>Y
61 POKEB
62 RCL Z
63 A+
64 PEEKB
65 R↑
66 A+
67 RTN
68 LBL 05
69 CLST
70 SEED
71 END

PLNG "PK"
113 BYTES
Further Hints

none

Related Functions

SAVEK, GETK, SAVEB, GETB
SORTFL

This function sorts the registers of a data file. The file name is specified in ALPHA. If the ALPHA register is empty, the current file is employed. Numeric as well as ALPHA data is sorted. The stack registers are not changed. After executing SORTFL, the first data register of the file now contains the smallest value of the specified data file.

Input

ALPHA register: file name

Further Hints

Contrary to the function SORT, descending sorting is not possible!

Related Function

SORT
At the end of this function block we will show just one more program which solves an often occurring problem quite easily:

**Storing a Text File on Magnetic Cards**

The program "WF/RF", which only contains one synthetic step, can be used for any extended memory file type, but was actually only written for text files (ASCII files). Line 18 appends six spaces to the ALPHA register. The second header register of the file must reside in the the Extended Functions module (absolute address 69 to 190).

The program "WF" (Write File) transfers all registers of the file to data registers. For this the number of data registers is set to the number of file registers (FLSIZE, PSIZE). Therefore this program can only be used for files which can be wholly saved into data registers (plus existing free registers). Nevertheless, files up to a size of over 300 registers can be processed in an HP-41 CV or HP-41 CX!

**Program use:**

The file name is specified in ALPHA. Now "WF" must be executed, to store the file on magnetic cards. At the input prompting "RDY 01 OF NN" it is imperative to slide in one (several) magnet card(s) or to start the program again using R/S twice. Otherwise the file type, which was temporarily changed, will not be changed back to the original file type.

Before being able to read back the file content from the magnetic cards using the program "RF" (Read File), the desired file must be constructed with the right file type and in sufficient length. Now the file name must again be placed in the ALPHA register and "RF" must be executed. After the input prompting "CARD" it is imperative to slide in one (several) magnet card(s) or to start the program again using R/S twice (see above).

The program can also just be used to transfer the file to data registers or vice versa. For this the functions WDTA and RDTA must be deleted. Now it is possible to enlarge a textfile without having to input the contents again. For this application one would execute "WF" and then destroy the file in extended memory and resize it larger, and then execute "RF".

(C) Copyright W&W Software Products GmbH 8.16
The program "WF/RF" can be used for all file types. Naturally, using it on program files is senseless. If the program is used for buffer files, the buffer file where the data will be saved must be of the same type as the buffer saved in the data registers. To avoid problems, the size should be the same as well. If all these restrictions are followed, there should be no problem storing all kinds of buffers on magnetic cards using "WF" and "RF".

```
01*LBL "WF"
02 XEQ 00
03 GETR
04 WDTA
05 POKEB
06 RTN

07*LBL "RF"
08 XEQ 00
09 RDTA
10 SAYER
11 POKEB
12 RTN

13*LBL 00
14 RCLPTA
15 FLSIZE
16 PSIZE
17 191.6
18 "I"
19 7
20 AROT
21 RCL [1
22 CLA
23*LBL 01
24 R↑
25 R↑
26 PEEKR
27 DSE Y
28 CLX
29 R↑
30 X≠Y?
31 GTO 01
32 RCL Z
33 PEEKB
34 X<>Y
35 32
36 POKEB
37 CLX
38 SEEKPT
39 X<> Z
40 TONE 8
41 END
```

PLNG "WF"
89 BYTES

(C) Copyright W&W Software Products GmbH 8.17
Chapter 9

Bar Codes
## Contents Chapter 9

### Bar Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A?&quot;</td>
<td>7.25</td>
</tr>
<tr>
<td>&quot;ABIN&quot;</td>
<td>4.77</td>
</tr>
<tr>
<td>&quot;BS?&quot;</td>
<td>8.08</td>
</tr>
<tr>
<td>&quot;CB&quot;</td>
<td>7.36</td>
</tr>
<tr>
<td>&quot;CDE&quot;</td>
<td>7.15</td>
</tr>
<tr>
<td>&quot;CF55&quot;</td>
<td>5.50</td>
</tr>
<tr>
<td>&quot;CHECK&quot;</td>
<td>7.34</td>
</tr>
<tr>
<td>&quot;CLK&quot;</td>
<td>8.11</td>
</tr>
<tr>
<td>&quot;GE&quot;</td>
<td>7.27</td>
</tr>
<tr>
<td>&quot;H-O&quot;</td>
<td>6.11</td>
</tr>
<tr>
<td>&quot;INP&quot;</td>
<td>6.05</td>
</tr>
<tr>
<td>&quot;INV&quot;</td>
<td>4.83</td>
</tr>
<tr>
<td>&quot;PBC&quot;</td>
<td>7.40</td>
</tr>
<tr>
<td>&quot;PHINPT&quot;</td>
<td>6.07</td>
</tr>
<tr>
<td>&quot;PK&quot;</td>
<td>8.13</td>
</tr>
<tr>
<td>&quot;PR1&quot;</td>
<td>6.22</td>
</tr>
<tr>
<td>&quot;PR4&quot;</td>
<td>6.24</td>
</tr>
<tr>
<td>&quot;ST&quot;</td>
<td>7.38</td>
</tr>
<tr>
<td>&quot;TD&quot;</td>
<td>7.31</td>
</tr>
<tr>
<td>&quot;TLC&quot;</td>
<td>7.33</td>
</tr>
<tr>
<td>&quot;TLC1&quot;</td>
<td>7.33</td>
</tr>
<tr>
<td>&quot;TLC2&quot;</td>
<td>7.33</td>
</tr>
<tr>
<td>&quot;VB&quot;</td>
<td>7.26</td>
</tr>
<tr>
<td>&quot;VR&quot;</td>
<td>7.30</td>
</tr>
<tr>
<td>&quot;W?&quot;</td>
<td>5.49</td>
</tr>
<tr>
<td>&quot;WF&quot;</td>
<td>8.17</td>
</tr>
</tbody>
</table>

Bar Codes of the Functions of the CCD-Module

(C) Copyright W&W Software Products GmbH

9.03
A?

1 (1-4)
2 (4-12)
3 (13-21)
4 (22-27)

ABIN

1 (1-2)
2 (2-11)
3 (11-15)
4 (16-20)
5 (20-28)
6 (29-35)
7 (36-40)
8 (41-47)
9 (47-53)
10 (54-62)
11 (63-71)
GE

1 (1-5)

2 (6-15)

3 (15-20)

H-0

1 (1-4)

2 (5-6)

3 (9-12)

4 (12-17)

5 (18-19)

INP

1 (1-4)

2 (4-11)

3 (12-19)

4 (20-26)
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1-5)</td>
</tr>
<tr>
<td>2</td>
<td>(6-13)</td>
</tr>
<tr>
<td>3</td>
<td>(13-20)</td>
</tr>
<tr>
<td>4</td>
<td>(21-29)</td>
</tr>
<tr>
<td>5</td>
<td>(30-41)</td>
</tr>
<tr>
<td>6</td>
<td>(42-49)</td>
</tr>
<tr>
<td>7</td>
<td>(50-60)</td>
</tr>
<tr>
<td>8</td>
<td>(60-68)</td>
</tr>
<tr>
<td>9</td>
<td>(69-78)</td>
</tr>
<tr>
<td>10</td>
<td>(79-85)</td>
</tr>
<tr>
<td>11</td>
<td>(85-94)</td>
</tr>
<tr>
<td>12</td>
<td>(95-103)</td>
</tr>
<tr>
<td>13</td>
<td>(104-112)</td>
</tr>
<tr>
<td>14</td>
<td>(112-118)</td>
</tr>
<tr>
<td>15</td>
<td>(118-125)</td>
</tr>
<tr>
<td>16</td>
<td>(125-135)</td>
</tr>
<tr>
<td>17</td>
<td>(136-143)</td>
</tr>
<tr>
<td>18</td>
<td>(144-150)</td>
</tr>
<tr>
<td>19</td>
<td>(151-156)</td>
</tr>
</tbody>
</table>
### VR

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1-5)</td>
</tr>
<tr>
<td>2</td>
<td>(5-11)</td>
</tr>
<tr>
<td>3</td>
<td>(12-16)</td>
</tr>
</tbody>
</table>

### W?

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1-6)</td>
</tr>
<tr>
<td>2</td>
<td>(7-11)</td>
</tr>
</tbody>
</table>

### WF

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1-4)</td>
</tr>
<tr>
<td>2</td>
<td>(5-9)</td>
</tr>
<tr>
<td>3</td>
<td>(9-16)</td>
</tr>
<tr>
<td>4</td>
<td>(17-18)</td>
</tr>
<tr>
<td>5</td>
<td>(19-27)</td>
</tr>
<tr>
<td>6</td>
<td>(28-36)</td>
</tr>
<tr>
<td>7</td>
<td>(36-43)</td>
</tr>
<tr>
<td>B?</td>
<td>CAS</td>
</tr>
<tr>
<td>----</td>
<td>-----</td>
</tr>
<tr>
<td>&gt;C+</td>
<td>&gt;R+</td>
</tr>
<tr>
<td>CMAXAB</td>
<td>CNRM</td>
</tr>
<tr>
<td>M+</td>
<td>M—</td>
</tr>
<tr>
<td>MDIM</td>
<td>MIN</td>
</tr>
<tr>
<td>R++</td>
<td>R—=</td>
</tr>
<tr>
<td>SUMAB</td>
<td>SWAP</td>
</tr>
<tr>
<td>1CMP</td>
<td>2CMP</td>
</tr>
<tr>
<td>OR</td>
<td>R&lt;</td>
</tr>
<tr>
<td>WSIZE</td>
<td>XOR</td>
</tr>
<tr>
<td>-I/O FNS</td>
<td>ABSP</td>
</tr>
<tr>
<td>CLA—</td>
<td>F/E</td>
</tr>
<tr>
<td>PRL</td>
<td>VIEWH</td>
</tr>
<tr>
<td>-ADV FNS</td>
<td>A+</td>
</tr>
<tr>
<td>PC&gt;X</td>
<td>PEEKB</td>
</tr>
<tr>
<td>PPLNG</td>
<td>X&gt;PC</td>
</tr>
<tr>
<td>MRGK</td>
<td>SAVEB</td>
</tr>
</tbody>
</table>
Chapter 10

Function Index
Function Index

The following CAT 2 function list of the CCD-Module shows the XROM numbers, byte combinations and the stack requirements of all CCD-Module functions.

**Sign Description:**
- The stack is not changed
↑ The stack is lifted; a result value is entered in X.
↓ Stack drop.
L The original X value is entered into the LASTX register.

### W&W CCD A

<table>
<thead>
<tr>
<th>FCN-NAME</th>
<th>XROM-#</th>
<th>BYTES</th>
<th>STACK</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>-W&amp;W CCD A</td>
<td>09.00</td>
<td>162=064</td>
<td>-</td>
<td>3.05</td>
</tr>
<tr>
<td>B?</td>
<td>09.01</td>
<td>162=065</td>
<td>-</td>
<td>3.09</td>
</tr>
<tr>
<td>CAS</td>
<td>09.02</td>
<td>162=066</td>
<td>-</td>
<td>3.07</td>
</tr>
<tr>
<td>CLB</td>
<td>09.03</td>
<td>162=067</td>
<td>↑</td>
<td>3.10</td>
</tr>
<tr>
<td>RNOM</td>
<td>09.04</td>
<td>162=068</td>
<td>-</td>
<td>3.08</td>
</tr>
<tr>
<td>SAS</td>
<td>09.05</td>
<td>162=069</td>
<td>-</td>
<td>3.12</td>
</tr>
<tr>
<td>SEED</td>
<td>09.06</td>
<td>162=070</td>
<td>-</td>
<td>3.13</td>
</tr>
<tr>
<td>SORT</td>
<td>09.07</td>
<td>162=071</td>
<td>-</td>
<td>3.13</td>
</tr>
</tbody>
</table>

### -ARR FNS

<table>
<thead>
<tr>
<th>FCN-NAME</th>
<th>XROM-#</th>
<th>BYTES</th>
<th>STACK</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ARR FNS</td>
<td>09.08</td>
<td>162=072</td>
<td>-</td>
<td>4.19</td>
</tr>
<tr>
<td>&gt;C+</td>
<td>09.09</td>
<td>162=073</td>
<td>-</td>
<td>4.21</td>
</tr>
<tr>
<td>&gt;R+</td>
<td>09.10</td>
<td>162=074</td>
<td>↑</td>
<td>4.14</td>
</tr>
<tr>
<td>?IJ</td>
<td>09.11</td>
<td>162=075</td>
<td>↑</td>
<td>4.15</td>
</tr>
<tr>
<td>?IJA</td>
<td>09.12</td>
<td>162=076</td>
<td>-</td>
<td>4.31</td>
</tr>
<tr>
<td>C&gt;C</td>
<td>09.13</td>
<td>162=077</td>
<td>↑</td>
<td>4.23</td>
</tr>
<tr>
<td>C&gt;+</td>
<td>09.14</td>
<td>162=078</td>
<td>↑</td>
<td>4.25</td>
</tr>
<tr>
<td>C&gt;-</td>
<td>09.15</td>
<td>162=079</td>
<td>↑</td>
<td>4.25</td>
</tr>
<tr>
<td>FCH-NAME</td>
<td>XROM-H</td>
<td>BYTES</td>
<td>STACK</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>CMAXAB</td>
<td>09.16</td>
<td>162:080</td>
<td>↑</td>
<td>4.43</td>
</tr>
<tr>
<td>CNRM</td>
<td>09.17</td>
<td>162:081</td>
<td>↑</td>
<td>4.54</td>
</tr>
<tr>
<td>CSUM</td>
<td>09.18</td>
<td>162:082</td>
<td>–</td>
<td>4.52</td>
</tr>
<tr>
<td>DIA</td>
<td>09.19</td>
<td>162:083</td>
<td>↑</td>
<td>4.13</td>
</tr>
<tr>
<td>FNRM</td>
<td>09.20</td>
<td>162:084</td>
<td>↑</td>
<td>4.56</td>
</tr>
<tr>
<td>IJ=</td>
<td>09.21</td>
<td>162:085</td>
<td>–</td>
<td>4.16</td>
</tr>
<tr>
<td>IJ=A</td>
<td>09.22</td>
<td>162:086</td>
<td>–</td>
<td>4.17</td>
</tr>
<tr>
<td>M+</td>
<td>09.23</td>
<td>162:087</td>
<td>–</td>
<td>4.57</td>
</tr>
<tr>
<td>M-</td>
<td>09.24</td>
<td>162:088</td>
<td>–</td>
<td>4.59</td>
</tr>
<tr>
<td>M*</td>
<td>09.25</td>
<td>162:089</td>
<td>–</td>
<td>4.61</td>
</tr>
<tr>
<td>M%M</td>
<td>09.26</td>
<td>162:090</td>
<td>–</td>
<td>4.65</td>
</tr>
<tr>
<td>M/</td>
<td>09.27</td>
<td>162:091</td>
<td>–</td>
<td>4.63</td>
</tr>
<tr>
<td>MAX</td>
<td>09.28</td>
<td>162:092</td>
<td>↑</td>
<td>4.39</td>
</tr>
<tr>
<td>MAXAB</td>
<td>09.29</td>
<td>162:093</td>
<td>↑</td>
<td>4.41</td>
</tr>
<tr>
<td>MDIM</td>
<td>09.30</td>
<td>162:094</td>
<td>–</td>
<td>4.09</td>
</tr>
<tr>
<td>MIN</td>
<td>09.31</td>
<td>162:095</td>
<td>–</td>
<td>4.45</td>
</tr>
<tr>
<td>MOVE</td>
<td>09.32</td>
<td>162:096</td>
<td>–</td>
<td>4.35</td>
</tr>
<tr>
<td>PIU</td>
<td>09.33</td>
<td>162:097</td>
<td>↑</td>
<td>4.46</td>
</tr>
<tr>
<td>R–PR</td>
<td>09.34</td>
<td>162:098</td>
<td>–</td>
<td>4.73</td>
</tr>
<tr>
<td>R–QR</td>
<td>09.35</td>
<td>162:099</td>
<td>–</td>
<td>4.69</td>
</tr>
<tr>
<td>R&lt;&gt;&lt;R</td>
<td>09.36</td>
<td>162:100</td>
<td>–</td>
<td>4.33</td>
</tr>
<tr>
<td>R&gt;</td>
<td>09.37</td>
<td>162:101</td>
<td>↑</td>
<td>4.27</td>
</tr>
<tr>
<td>R&gt;–</td>
<td>09.38</td>
<td>162:102</td>
<td>↑</td>
<td>4.29</td>
</tr>
<tr>
<td>R&gt;&gt;&lt;R?</td>
<td>09.39</td>
<td>162:103</td>
<td>–</td>
<td>4.47</td>
</tr>
<tr>
<td>RMAXAB</td>
<td>09.40</td>
<td>162:104</td>
<td>↑</td>
<td>4.44</td>
</tr>
<tr>
<td>RNRN</td>
<td>09.41</td>
<td>162:105</td>
<td>↑</td>
<td>4.55</td>
</tr>
<tr>
<td>RSUM</td>
<td>09.42</td>
<td>162:106</td>
<td>–</td>
<td>4.53</td>
</tr>
<tr>
<td>SUN</td>
<td>09.43</td>
<td>162:107</td>
<td>↑</td>
<td>4.49</td>
</tr>
<tr>
<td>SUMAB</td>
<td>09.44</td>
<td>162:108</td>
<td>↑</td>
<td>4.51</td>
</tr>
<tr>
<td>SWAP</td>
<td>09.45</td>
<td>162:109</td>
<td>–</td>
<td>4.37</td>
</tr>
<tr>
<td>YC+C</td>
<td>09.46</td>
<td>162:110</td>
<td>–</td>
<td>4.67</td>
</tr>
</tbody>
</table>

-HEX FNS

<table>
<thead>
<tr>
<th>-HEX FNS</th>
<th>XROM-H</th>
<th>BYTES</th>
<th>STACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1CMP</td>
<td>09.47</td>
<td>162:111</td>
<td>L</td>
</tr>
<tr>
<td>2CMP</td>
<td>09.48</td>
<td>162:112</td>
<td>L</td>
</tr>
<tr>
<td>AND</td>
<td>09.49</td>
<td>162:113</td>
<td>L</td>
</tr>
<tr>
<td>AND</td>
<td>09.50</td>
<td>162:114</td>
<td>L</td>
</tr>
</tbody>
</table>

(C) Copyright W&W Software Products GmbH 10.04
<table>
<thead>
<tr>
<th>FCN-NAME</th>
<th>XROM-#</th>
<th>BYTES</th>
<th>STACK</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A+</td>
<td>11.18</td>
<td>162:210</td>
<td>L</td>
<td>7.09</td>
</tr>
<tr>
<td>A+B</td>
<td>11.19</td>
<td>162:211</td>
<td>↓L</td>
<td>7.11</td>
</tr>
<tr>
<td>A-</td>
<td>11.20</td>
<td>162:212</td>
<td>L</td>
<td>7.12</td>
</tr>
<tr>
<td>DCD</td>
<td>11.22</td>
<td>162:214</td>
<td></td>
<td>7.14</td>
</tr>
<tr>
<td>PC&lt;&gt;RTN</td>
<td>11.23</td>
<td>162:215</td>
<td></td>
<td>7.21</td>
</tr>
<tr>
<td>PC&gt;X</td>
<td>11.24</td>
<td>162:216</td>
<td>↑</td>
<td>7.16</td>
</tr>
<tr>
<td>PEEKB</td>
<td>11.25</td>
<td>162:217</td>
<td>↑</td>
<td>7.24</td>
</tr>
<tr>
<td>PEEKR</td>
<td>11.26</td>
<td>162:218</td>
<td>↑</td>
<td>7.29</td>
</tr>
<tr>
<td>PHD</td>
<td>11.27</td>
<td>162:219</td>
<td>↑</td>
<td>7.08</td>
</tr>
<tr>
<td>PLNG</td>
<td>11.28</td>
<td>162:220</td>
<td>↑</td>
<td>7.05</td>
</tr>
<tr>
<td>POKEB</td>
<td>11.29</td>
<td>162:221</td>
<td></td>
<td>7.31</td>
</tr>
<tr>
<td>POKER</td>
<td>11.30</td>
<td>162:222</td>
<td></td>
<td>7.35</td>
</tr>
<tr>
<td>PPLNG</td>
<td>11.31</td>
<td>162:223</td>
<td>↑</td>
<td>7.07</td>
</tr>
<tr>
<td>X&gt;PC</td>
<td>11.32</td>
<td>162:224</td>
<td></td>
<td>7.18</td>
</tr>
<tr>
<td>X&gt;RTN</td>
<td>11.33</td>
<td>162:225</td>
<td></td>
<td>7.20</td>
</tr>
<tr>
<td>XR&gt;RTN</td>
<td>11.34</td>
<td>162:226</td>
<td></td>
<td>7.23</td>
</tr>
</tbody>
</table>

-XF/M FNS

<table>
<thead>
<tr>
<th>FCN-NAME</th>
<th>XROM-#</th>
<th>BYTES</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GETB</td>
<td>11.36</td>
<td>162:228</td>
<td>8.07</td>
</tr>
<tr>
<td>GETK</td>
<td>11.37</td>
<td>162:229</td>
<td>8.10</td>
</tr>
<tr>
<td>MRGK</td>
<td>11.38</td>
<td>162:230</td>
<td>8.12</td>
</tr>
<tr>
<td>SAVEB</td>
<td>11.39</td>
<td>162:231</td>
<td>8.05</td>
</tr>
<tr>
<td>SAVEK</td>
<td>11.40</td>
<td>162:232</td>
<td>8.09</td>
</tr>
<tr>
<td>SORTFL</td>
<td>11.41</td>
<td>162:233</td>
<td>8.15</td>
</tr>
<tr>
<td>FCN—NAME</td>
<td>XROM—#</td>
<td>BYTES</td>
<td>STACK</td>
</tr>
<tr>
<td>--------------</td>
<td>--------</td>
<td>-------------</td>
<td>-------</td>
</tr>
<tr>
<td>A+</td>
<td>11.18</td>
<td>162=210</td>
<td>L</td>
</tr>
<tr>
<td>A+B</td>
<td>11.19</td>
<td>162=211</td>
<td>↓L</td>
</tr>
<tr>
<td>A—</td>
<td>11.20</td>
<td>162=212</td>
<td>L</td>
</tr>
<tr>
<td>A−A</td>
<td>11.21</td>
<td>162=213</td>
<td>↓L</td>
</tr>
<tr>
<td>DCD</td>
<td>11.22</td>
<td>162=214</td>
<td>—</td>
</tr>
<tr>
<td>PC&lt;&gt;RTN</td>
<td>11.23</td>
<td>162=215</td>
<td>—</td>
</tr>
<tr>
<td>PC&gt;X</td>
<td>11.24</td>
<td>162=216</td>
<td>↑</td>
</tr>
<tr>
<td>PEEKB</td>
<td>11.25</td>
<td>162=217</td>
<td>↑</td>
</tr>
<tr>
<td>PEEKR</td>
<td>11.26</td>
<td>162=218</td>
<td>↑</td>
</tr>
<tr>
<td>PHD</td>
<td>11.27</td>
<td>162=219</td>
<td>↑</td>
</tr>
<tr>
<td>PLNG</td>
<td>11.28</td>
<td>162=220</td>
<td>↑</td>
</tr>
<tr>
<td>POKEB</td>
<td>11.29</td>
<td>162=221</td>
<td>—</td>
</tr>
<tr>
<td>POKER</td>
<td>11.30</td>
<td>162=222</td>
<td>—</td>
</tr>
<tr>
<td>PPLNG</td>
<td>11.31</td>
<td>162=223</td>
<td>↑</td>
</tr>
<tr>
<td>X&gt;PC</td>
<td>11.32</td>
<td>162=224</td>
<td>—</td>
</tr>
<tr>
<td>X&gt;RTN</td>
<td>11.33</td>
<td>162=225</td>
<td>—</td>
</tr>
<tr>
<td>XR&gt;RTN</td>
<td>11.34</td>
<td>162=226</td>
<td>—</td>
</tr>
</tbody>
</table>

**-XF/M FNS**

<table>
<thead>
<tr>
<th>-XF/M FNS</th>
<th></th>
<th>162=227</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GETB</td>
<td>11.36</td>
<td>162=228</td>
<td>—</td>
<td>8.07</td>
</tr>
<tr>
<td>GETK</td>
<td>11.37</td>
<td>162=229</td>
<td>—</td>
<td>8.10</td>
</tr>
<tr>
<td>MRGK</td>
<td>11.38</td>
<td>162=230</td>
<td>—</td>
<td>8.12</td>
</tr>
<tr>
<td>SAVEB</td>
<td>11.39</td>
<td>162=231</td>
<td>—</td>
<td>8.05</td>
</tr>
<tr>
<td>SAVEK</td>
<td>11.40</td>
<td>162=232</td>
<td>—</td>
<td>8.09</td>
</tr>
<tr>
<td>SORTFL</td>
<td>11.41</td>
<td>162=233</td>
<td>—</td>
<td>8.15</td>
</tr>
</tbody>
</table>
Chapter 11

Compatibility
Compatibility

Once you have plugged the CCD-Module into your calculator, it may happen, as described below, that some functions do work as described. To avoid this, please check if any of the points mentioned below apply to you:

1) Your calculator has an old operating system. The operating system extensions are only possible, if the module can, after every press of a key, take control of the calculator for a short while. This is only possible with the newer operating system of the HP-41. Therefore, if you possess an older HP-41 (i.e., the serial number is smaller than 2035...), it may be possible that some CCD-Module operating system extensions will not work. If this is the case you can:

a) simulate some functions, by plugging the CCD-Module into Port 1 and reading the barcodes for "ASN", "CAT" and "XEQ" (see barcodes).

ASN

<table>
<thead>
<tr>
<th>1</th>
<th>(1-3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>(4-11)</td>
</tr>
<tr>
<td>3</td>
<td>(11-19)</td>
</tr>
<tr>
<td>4</td>
<td>(20)</td>
</tr>
</tbody>
</table>

CAT

<table>
<thead>
<tr>
<th>1</th>
<th>(1-4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>(4-7)</td>
</tr>
</tbody>
</table>
b) Suppose a barcode reader is plugged in, press a key, if "Prompt" is displayed. Through this the CCD-Module takes control and the extended function is put to use.

c) Exchange your operating system against the usual costs at any HP service center

For incompatibilities with possibly newly developed HP-41 models no liability is taken. The CCD-Module works with all HP-41 models built until January 1985.

2) You work the CCD-Module together with the HP-IL development module. If you would like to use an HP-development module in conjunction with the CCD-Module, make sure that the CCD-Module is before the HP-IL development module. If this is not the case, some of the CCD-Module operating system extensions may not work. If the modules are plugged in the order explained above, there is nothing to worry about.

3) Using the CCD-Module in connection with the ZENROM. When employing the ZENROM, there is an incompatibility in the small letter mode, since the ZENROM is able to produce small letters as well. This can only be avoided by switching off the small letter mode of the CCD-Module (see program "TLC").

4) Using the CCD-Module in conjunction with a module of the same XROM number. Once you have plugged in the CCD-Module, further modules with the XROM numbers 9 or 11 may not be plugged in anymore.

Up until now these are the following:

- Home management module
- Real estate module
- PANAME module
Chapter 12

Literature
Literary Hints

If you are interested in knowing more about synthetic or optimized programming of your HP-41, we recommend the following books:

1) Synthetic Programming on the HP-41C
   Author: W. C. Wickes

2) HP-41 Synthetic programming made easy
   Author: K. Jarrett

3) Optimales Programmieren mit dem HP-41
   Author: Gerhard Kruse
NOTICE

W&W Software Products makes no express or implied warranty with regard to the program material offered or the merchantibility or the fitness of the program material for any particular purpose. The program material is made available solely on an "as is" basis, and the entire risk as to its quality and performance is with the user. Should the program material prove defective, the user (and not W&W Software Products nor any other party) shall bear the entire cost of all necessary correction and all incidental or consequential damages. W&W Software Products shall not be liable for any incidental or consequential damages in connection with or arising out of the furnishing, use, or performance of the program material.