WP 43S OWNER'S MANUAL

This manual documents *WP 43S*, a free scientific software for the calculator *DM42* of *SwissMicros*. You can redistribute *WP 43S* and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

WP 43S is published and distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. Please see the GNU General Public License at http://www.gnu.org/licenses/ for more details.

This manual is very preliminary; it will change while we develop *WP 43S* in course of this project. We reserve the right to do so at any time. The very basic principles of *WP 43S* will stay constant, however. Stay informed by watching https://gitlab.com/Over_score/wp43s .

 For the time being, the locations highlighted cyan are open construction sites – information is missing there or needs further discussion and investigation to be determined. Any contributions in this matter are highly appreciated.

The pictures on pp. 53, 200f, and 321, the stack graphics on pp. 33 (top), 34 (top), and 37, as well as the paintings and drawings on pp. 22, 45, 50, 65, 82f, 88f, 95-97, 105- 108, 119, 129-132, 155, 158-160, 205, 213, 237, 240, 248, and in Appendix 3 were taken from various calculator manuals and advertisements published by *Hewlett-Packard* between 1974 and 1989. The diagrams on p. 98 are based on material found in *Wikipedia*. All *WP 43S* keyboard graphics as well as the other photographs, pictures, graphics, and diagrams printed in this manual were created by the author.

Internet addresses are specified as found and verified at 2019-06-26 (just the map printed on p. 92 is not found online anymore). Please note such addresses may change without notice at any time.

This manual is published in English since it became the *lingua franca* of our time (after Greek, Latin, and French) – using it we can reach the maximum number of people without further translations. I apologize to the people of other languages and inserted some 'translator's notes' where applicable.

Printed in the USA

ISBN-13: 978-172950098-9
ISBN-10: 172950098-6

WP 43S would not have been created without our love for *Classics, Woodstocks, Stings, Spices, Nuts, Voyagers*, and *Pioneers*. Thus we want to quote what was printed in *Hewlett-Packard* pocket calculator manuals until 1980, so it will not fade:

"The success and prosperity of our company will be assured only if we offer our customers superior products that fill real needs and provide lasting value, and that are supported by a wide variety of useful services, both before and after sales."

Statement of Corporate Objectives
Hewlett-Packard

TABLE OF CONTENTS

Welcome! 8

Print Conventions and Common Abbreviations 12

Section 1: Getting Started 15

Problem Solving, Part 1: First Steps 18

How to Enter Common Numbers (and How to Edit Them) 25

How to Enter and Execute Commands 26

Menus – Items à la carte 27

How the Keyboard is Organized 29

Problem Solving, Part 2: Elementary Stack Mechanics 31

Looking Closer at the Automatic Stack 38

Problem Solving, Part 3: The Stack in Advanced Calculations 41

Special Tricks, #1: Top Stack Level Repetition 48

Special Tricks, #2: LASTx for Reusing Numbers 50

Error Recovery: [←], [EXIT], and [↶] 51

Clearing and Resetting Your WP 43S 52

Addressing and Manipulating Objects in RAM 53

Addressing Tables 60

Indirect Addressing – Working with Pointers 64

Store and Recall Arithmetic 64

Section 2: Dealing with Various Objects and Data Types 67

Some Display Basics 67

Supported Data Types 68

Recognizing Calculator Settings and Status 74

Getting Special Information: RBR, STATUS, VERS, etc. 77

Localising Numeric Output 78

Real Numbers: Changing the Display Format 80

Real Numbers: Squares and Cubes and their Roots 82

Real Numbers: Percent Change 85

Real Numbers: Logarithms and Powers (a.k.a. Antilogs) 86

Real Numbers: Hyperbolic Functions 94
Real Numbers: Probabilities – Factorials, Combinations, Permutations, and Distributions 96
Real Numbers: From Probability to Statistics – Accumulating Data, Calculating Means, Standard Deviations, and Confidence Limits; Curve Fitting, Forecasting, and Checking Dices 99
Real Numbers: Some Industrial Problems Solved 109
Real Numbers: Summary of Functions 118

Angles and Trigonometric Functions 125
Mixed Calculations: Coordinate Transformations in 2D, Flight Directions, Courses over Ground, etc. 128
Angles: Summary of Functions 134

Integers: Input and Displaying 135
Integers: Bitwise Operations on Short Integers 140
Integers: Arithmetic Operations 143
Integers: Overflow and Carry with Short Integers 145
Integers: Summary of Functions 148

Rational Numbers (Fractions) 151

Complex Numbers: Introduction 154
Complex Numbers Used for 2D Vector Algebra 158
Complex Numbers: Summary of Functions 161

Vectors and Matrices: Introduction and Input 163
Vectors and Matrices: Displaying and Editing Larger Objects 168
Vectors and Matrices: Complex Stuff 172
Vectors and Matrices: Calculating 174
Vectors and Matrices: Solving Systems of Linear Equations 179
Vectors and Matrices: Eigenvalues and Eigenvectors 180
Vectors and Matrices: Dealing with Statistical Data 185
Vectors and Matrices: Summary of Functions 187

Times 189

Dates 191

Alpha Input Mode: Introduction and Virtual Keyboard 193
Alpha Input Mode: Entering Simple Text and More 195
Combining Alpha Strings and Numeric Data 197
Working with Alphanumeric Strings 198

Section 3: Programming 202
Recording a New Routine 204
Labels 208
Editing a Routine 211
Running a Routine from the Keyboard (also for Debugging) 212
Subroutines: Running a Routine from another Routine 213
Automatic Testing and Conditional Branching 214
Loops and Counters 218
Programmed User Interaction and Dialogues 221
Solving Differential Equations 225
The Programmable Menu (MENU) 230
Basic Kinds of Program Steps 231
Deleting Programs 232
Serial Input and Output of Data and Programs 233
Local Data 233
Flash Memory (*FM*) 234

Section 4: Advanced Problem Solving 235
Programmable Sums 235
Programmable Products 236
Solving Quadratic Equations 237
General Equations 238
The Interactive Solver for Arbitrary Equations 240
The Interactive Solver for Expressions Stored in Programs 244
Using the Solver in a Program 246
Numeric Integration of Equations 248
Interactively Integrating Expressions Stored in Programs 251
Using the Integrator in a Program 252
Differentiating Equations 254
Interactively Differentiating Expressions Stored in Programs 256
Computing Derivatives in a Program 257

Nesting Advanced Operations 258

Section 5: Two Browsers, two Applications, and two Special Menus 261

The Browsers RBR and STATUS 261

The Timer Application 264

The Time Value of Money (TVM) 266

Constants 270

Unit Conversions 276

Section 6: Creating Your Very Personal *WP 43S* 281

Assigning Your Favourite Functions 283

Creating Your Own Menus 287

Browsing and Purging Menus, Variables, and Programs 290

Assigning Special Characters 290

User Mode 292

Appendix 1: Operator Precedence 295

Appendix 2: Key Response Table 296

Appendix 3: Further Applications of TVM 310

Ordinary Annuities (a.k.a. Payments in Arrears) 311

Annuities Due (a.k.a. Payments in Advance) 315

Appendix 4: Power Supply 319

Appendix 5: Time Line of Quoted Manuals 320

Appendix 6: Release Notes 322

Index 326

WELCOME!

Congratulations, you are holding your very own *WP 43S* in your hands now. It is a true pioneer: the **very first entirely community-designed and -built *RPN* pocket calculator**.[1]

All the hardware, firmware, and user interface of your *WP 43S* were thoroughly thought through, discussed, designed and assembled, written and tested by us over and over again. We did this to create a **new fast, compact, and solid problem solver like you did never own before** – instant on, fully programmable, incorporating a state-of-the-art LCD, customizable by you, still comfortably fitting into your shirt pocket, and *RPN* – **a serious scientific instrument** supporting you in your professional activities. It readily provides several advanced capabilities never before combined so conveniently in a pocket calculator:

- A *Solver* (root[2] finder) that can solve for any variable in an arbitrary equation.
- A numeric integrator for computing definite integrals of arbitrary functions.
- Numeric derivation, programmable sums and products.
- A wealth of functions, operating on real and complex numbers and matrices, integers, fractions, dates, times, and text strings.
- A comprehensive set of statistical operations including probability distributions, data analysis functions, curve fitting, and forecasting.
- Matrix operations including a comfortable *Matrix Editor*, a solver for systems of linear equations, eigenvalues, eigenvectors, and many more matrix and vector functions in real and complex domain.

---

[1] *RPN* stands for *Reverse Polish Notation*, a very effective and coherent method for most efficient solutions to complicated problems. It is based on a mathematical logic known as *Polish Notation* (since invented by the Polish logician *Jan Łukasiewicz,* see *https://www.youtube.com/watch?v=qRrAj-GCTQM*). He placed operators before numbers or variables instead between them as in conventional mathematical notation. Thus, *RPN* places operator behind numbers. See *Section 1* of this manual for more.

Our hardware platform, the *DM42* of *SwissMicros*, was developed in parallel and launched in 2017. This is due to the fact that the layout of the *DM42* is very closely linked to the *HP-42S* of *Hewlett-Packard* produced until 1995 and its firmware uses *Thomas Okken's Free42* simulating *HP-42S*.

[2] Translator's note for German readers: *Root* bedeutet hier *Nullstelle*.

- Base conversions and integer arithmetic in fifteen bases from binary to hexadecimal. Bit manipulations in *words* of up to 64 *bits*.
- A timer based on a real-time clock.
- An easy-to-use *menu* system using the bottom part of the display to assign up to eighteen operations to the top six keys according to your actual needs.
- Keystroke programming including branching, looping, tests, *flags*, subroutines, and program-specific local data.
- Easy system control via named *system flags* and variables provided.
- A *catalog* for reviewing all *items* stored in memory – be they provided by us or defined and programmed by you.
- A keyboard layout and *menus* you can customize. You can save your various custom layouts on-board and recall them one by one as you need them. Keyboard overlays are supported.
- Battery-fail-safe on-board backup memory for all your data (in *registers*, variables, *menus*, programs, layouts, and mode settings).
- A micro *USB* socket allowing for external auxiliary power supply as well as for exchanging your programs with a computer, so you can edit, debug, and test them there and return them proven thereafter.
- An infrared port for immediate recording of results, calculations, programs, and data using e.g. an *HP 82240A/B Infrared Printer*.

Your *WP 43S* provides the most ample function set ever seen in an *RPN* pocket calculator, presumably in any pocket calculator at all:

- More than 650 functions, including *Euler's Beta* and *Riemann's Zeta*, *Lambert's W*, the *error function*, *Bessel functions* of first kind, *Bernoulli* and *Fibonacci numbers*, as well as the *Chebyshev*, *Hermite*, *Laguerre*, and *Legendre* orthogonal polynomials – no more need for carrying heavy printed tables or running computer software for this matter.
- 14 probability distributions: general *Normal*, *Student's t*, *chi-square*, *Fisher's F*, *Poisson*, *binomial*, *geometric*, *hypergeometric*, *Cauchy-Lorentz*, *exponential*, *logistic*, *Weibull*, and more.
- 10 curve fitting models with two or three parameters (two kinds of linear, exponential, logarithmic, power, root, hyperbolic, parabolic, Cauchy and Gauss peak).

- Over 50 fundamental physical constants as accurate as used today by national standards institutes such as *NIST* or *PTB* (following CODATA 2018), plus a set of important mathematical, astronomical, and surveying constants.
- More than 100 unit conversions, mainly from old *British Imperial* to universal *SI* units and vice versa.
- Plus a complete set of financial functions and applications for the inevitable business matters.

Furthermore, your *WP 43S* features lots of space for your data, programs, and ideas:

- A high-resolution low-power dot-matrix *LCD* (240 × 400 pixels), showing crisp results and *menus*, allowing for natural matrix display as well as for a wide variety of mathematical symbols, Greek and extended Latin letters.
- Your choice of 4 or 8 *stack registers* and up to 107 global general purpose *registers*, each taking any object of arbitrary *data type* (be it a matrix, a vector, a string, or any number of arbitrary kind).
- Up to 1000 named variables. Also each such variable can take any object of arbitrary *data type*.
- 112 global *user flags* and 40 named *system flags*.
- 16 local *user flags* and up to 99 local *registers* per program, allowing e.g. for recursive programming.
- Up to 10 000 program steps in RAM, up to 20 000 program steps in *flash memory*.

WP 43S is the result of an international collaboration of two teams: *SwissMicros* (https://www.swissmicros.com/) – i.e. Swiss *Michael Steinmann* and Czech *David Jedelsky* – created the hardware, French *Martin Lorang*, German *Gert Menke* and *Friedrich Mütschele*, Italian *Gianluca Puggelli*, Dutch *Harald Overbeek*, South African *Jaco Mostert*, Australian *Paul Dale*, and me designed the user interface and wrote the software.[3]

[3] The firmware of your *WP 43S* is based to a large extent on the experience *Paul* and me gained with the *WP 34S RPN Scientific* calculator on the market since 2011. We started the *WP 34S* project in 2008. You find specific information about the *WP 34S*

As our *WP 34S* and *WP 31S* before, also *WP 43S* is a hobbyist's project. It started in public in 2012 and was presented and discussed on https://www.hpmuseum.org/forum/forum-8.html until June 2016 and on https://forum.swissmicros.com/ from 2017 on.[4] Prototypes of the *SwissMicros* hardware were publicly shown first on the *HHC2016* conference in Nashville (USA) and on the *Allschwil Meeting 2016* in Switzerland. *Martin* and me presented the first version of the *WP 43S* simulator on the *Allschwil Meeting 2018*. We thank the participants of said meetings and all members of the international community who contributed their ideas, put their votes, and lent their support at various phases throughout this project. We greatly appreciate your contributions!

We baptized our baby in honor of the *HP-42S* of 1988, the most powerful *RPN* pocket calculator available before these activities.[5] May it be a worthy and valiant successor of the *HP-42S* – though we would have preferred *Hewlett-Packard* making it (the company as we knew it until the 80's of last century). In any way, *WP 43S* stands in the tradition of almost 50 years of *RPN* pocket calculators.

We carefully checked all aspects of *WP 43S* to the best of our ability. Thus we hope it is free of severe bugs. This cannot be guaranteed, however, so we promise to continue improving *WP 43S* whenever necessary. Should you discover any strange results, please report them to us. If they turn out being caused by internal bugs, we will correct the firmware and provide you with an update as soon as possible. As we did since 2011, we will continue maintaining short response times.

Enjoy!

Walter Bonin

---

and its derivative, the *WP 31S*, at https://sourceforge.net/projects/wp34s/ and the links mentioned there. Both these calculators are based on *HP* hardware.

[4] If you are interested in the long and winding road how your *WP 43S* got the features, shape, and layout you're facing now, see the *Release Notes* in *App. 5* at the end of this manual and the two websites mentioned.

[5] For us, a pocket calculator per definition is a device fitting comfortably in your shirt pocket. Marketing people are observed to see this term more elastic – our shirt pockets are not elastic enough for that. Being at it, we generally recommend <u>not</u> to put your calculator in the back pocket of your jeans – it may break or multiply there.

Print Conventions and Common Abbreviations

- Throughout this manual, standard text font is Arial. Emphasis is added by underlining or **bold** printing. Calculator COMMANDS, MENUS, PREDEFINED VARIABLES and SYSTEM FLAGS are generally called by their names, printed capitalized in running text (*menus* underlined). Quoted text is printed blue (as well as translator's notes).

  Specific terms, titles, trademarks, names or *abbreviations* are printed in italics, *hyperlinks* in blue underlined italics. The latter will beam you to its target in the original `.pdf` file – it cannot work in a printed copy for obvious reasons; thus, such links generally refer to page numbers, to the *Table of Contents*, or to fully specified external addresses.

  Bold italic Arial letters such as ***n*** are employed for variables; bold regular letters for constant **sample values** (e.g. specific labels, numbers, or characters).

- Times New Roman regular letters are for unit symbols and for mathematical functions; italics are for *unit names* in running text.

  Times New Roman bold capitals are used for **REGISTER ADDRESSES**, lower case bold italics for ***register contents***. So e.g. the variable value ***y*** lives in *register* **Y** and ***r45*** in **R45**. Overall *stack* contents are generally quoted in the order [***x***, ***y***, ***z***, …]. We keep the term *register* for the space where an individual object is stored, although the actual size of such a *register* may vary widely following the size of the object stored therein.

- This KEY font (created by *Luiz Vieira* of Brasil) is taken for references to calculator keys, including SOFTKEYS in general. For shifted operations like GTO or LBL, the respective color is used. Alphanumeric and numeric calculator outputs (like 1.234×10^{-56} or $7{,}089\cdot10^{-12}$) are printed as you see them on the calculator screen.

- Courier is used for `file names` and describing numeric `formats`.

- Regarding mathematical symbols and notation, we generally follow *ISO 80000-2*. We use decimal points and multiplication crosses in most parts of this manual (but you may set your *WP 43S* to decimal commas and/or multiplication dots as well, of course). Although that point is less visible than a comma, 'comma people' seem to be more tolerant against decimal points than ‘point people’ against decimal commas (based on the number of complaints read so far).

All this holds unless stated otherwise locally.

The following abbreviations, listed alphabetically, are used throughout this manual – find detailed information about the respective terms at the locations referred to in the *Index* on pp. 326f, if applicable:

AIM	=	**a**lpha **i**nput **m**ode.
App.	=	**A**ppendix.
DT	=	**d**ata **t**ype.
GP	=	**g**eneral **p**urpose.
HP	=	***H**ewlett-**P**ackard.*
IOI	=	***I**ndex **o**f all **I**tems* provided in the *WP 43* (see *ReM* below).
LCD	=	**l**iquid **c**rystal **d**isplay.
OH	=	***O**wner’s **H**andbook.*
OHPG	=	***O**wner’s **H**andbook and **P**rogramming **G**uide.*
OM	=	***O**wner’s **M**anual.*
PEM	=	**p**rogram-**e**ntry **m**ode.
RAM	=	**r**andom **a**ccess **m**emory, allowing read and write operations.
ReM	=	*WP 43S **R**eference **M**anual*, containing also the *IOI.*
RPN	=	***R**everse **P**olish **N**otation* (cf. footnote 1 on p. 8).
SI	=	***s**ystème **i**nternational d’unités*, a coherent system of units of measurement agreed on internationally and adopted by almost all countries on this planet.[6]

Further abbreviations are listed in the *Index*. A few more may be used and explained locally.

[6] Only Liberia, Myanmar, and the USA are not participating yet. We do not know what they are afraid of – and they obviously do not know what they miss.

If you should need basic information about *SI* and its foundations, please turn to https://en.wikipedia.org/wiki/International_System_of_Units. See also the chapters about *Constants* and *Unit Conversions* in *Section 5* below.

Finally: **WARNING** indicates the risk of severe errors. There are only three warnings printed in this manual.[7] Resetting your calculator will help in almost all cases – but it will erase your data in *RAM*.

[7] … although *WP 43S* is distributed in the USA as well.

SECTION 1: GETTING STARTED

At its heart, your *WP 43S* is an extremely powerful, versatile problem-solving tool. It allows you to solve even very elaborate mathematical and computational problems in either of two different ways:

- **Manual problem solving:** Using the calculator's *RPN* logic system, you can manually work step-by-step through the toughest problems while seeing intermediate answers each and every step of the way. The advantages of *RPN* become particularly apparent when working with exploratory type problems where intermediate answers are an important part of the problem solving process.
- **Programmed problem solving:** Your *WP 43S* can remember any sequence of keystrokes you entered, and it can then run it repeatedly as often as you need it. This simple programming paradigm is particularly useful in providing answers to repetitive problems that require different inputs. Advanced programs may also be written for solving more elaborate tasks, e.g. iterative computations containing automatic decisions and branching. Thousands of keystrokes can be recorded in your *WP 43S* and can be exchanged with your computer or laptop.

If you know how to deal with a good old *HP RPN* scientific calculator, you can start using your *WP 43S* right away. Browse this manual to learn about some fundamental design concepts putting your *WP 43S* ahead of previous scientific pocket calculators.

On the other hand, if *RPN* is new to you, we recommend going through *Sections 1* and *2* of this manual thoroughly. This will enable you to easily solve problems manually benefitting from this unique logic system implemented. Once learned, *RPN* forms a lifelong lasting, reliable basis of your work.

Most commands on your *WP 43S* work as they did on its antecessors, in particular on the *WP 34S* and *HP-42S*. This manual is designed to supplement your prior knowledge, focussing on the new features of your *WP 43S* and providing information about them. It includes also some formulas and technical explanations; but it is not intended to replace text-books on mathematics, statistics, physics, engineering, economy, computer science, or programming.

The following text starts with presenting you the keyboard of your *WP 43S*, so you learn where to find what you are looking for. It continues demonstrating basic calculation methods, the memory of your *WP 43S* and addressing objects therein.

Section 2 covers the display and its indicators giving you feedback about what is going on. Furthermore, the various *data types* supported by your *WP 43S* are presented and demonstrated comprehensively.

Programming your *WP 43S* (as shown in *Section 3*) follows field proven concepts known from successful previous pocket calculators up to and including the *HP-42S* and *WP 34S*.

Sections 4 and *5* present advanced functionalities implemented in your *WP 43S*. You will find everything about the opportunity of customizing your *WP 43S* according to your very personal preferences in *Section 6*.

This *Owner's Manual* is supplemented by a *Reference Manual*. A major part of the latter is taken by the *IOI*, an index of all available operations (and more), what they do, and how to call them. It also contains full information about all the *menus* and *system flags* provided. The *ReM* closes covering special topics (e.g. memory management, a *WP 43S* emulator for your computer, and advanced mathematical functions implemented). There you also find instructions for keeping your *WP 43S* up-to-date whenever new firmware revisions will be released. Continue using both manuals for reference.

Before diving into the *OM*, here is something we ask you to remember:

> **Your *WP 43S* is designed to support you solving analytical problems. But it is just a mathematical tool (although a very powerful one): it can neither think for you nor check the sensibility of a problem you apply it to.**
>
> Thus, please do not blame us nor it for errors you may have made. Gather information, think before and while keying in and calculating, and check your results: these tasks will remain <u>your</u> responsibilities always. We will not be liable for any of your results.

SAFETY WARNING: Your *WP 43S* is <u>not</u> designed to be used by children under 3 years. This is <u>not</u> a toy. Do not use it before you can read. Your *WP 43S* contains small parts which if swallowed are

hazardous for health. Swallowing the battery can be fatal within 2 hours – seek medical advice immediately! [8]

Do not use your *WP 43S* for any other purposes than specified above (e.g. not as a hammer, a lever, a door stop, or a missile); **else you may destroy your *WP 43S* and / or other objects easily, or even hurt animals or human beings including yourself.** [8]

Do not drop your *WP 43S* on solid ground – it may break. [8]

Your *WP 43S* shall be operated in a clean and dry environment (relative humidity less than 35%) at ambient temperatures between 5 and 40 °C (41 to 104 °F).

Do not leave your *WP 43S* laying in the sun; its dark surface may become hot, and hot surfaces may cause burns. Do not leave your *WP 43S* laying in the cold; humidity may condense on its surface when a cold *WP 43S* is put in a warmer environment then.

Should your *WP 43S* become wet, turn it off, remove the battery (see *Disassembly* below), and let your *WP 43S* dry for sufficient time before turning it on again. Do not try to accelerate drying by blowing hot air (exceeding 60 °C or 140 °F) into your *WP 43S* or by putting it into a microwave oven or the like – you may destroy it. [8]

Disassembly: Do not disassemble your *WP 43S* unless you are qualified for such work and have the proper tools handy. You will need 1 small Phillips screwdriver and 2 hands for opening it. [8]

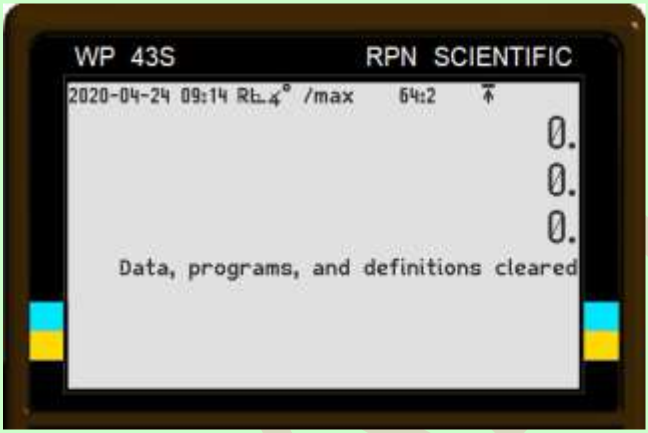
Inserting / replacing the battery: Your *WP 43S* contains a battery. When it runs out of power, 🔋 will appear top right on the screen. Then open your *WP 43S* (cf. *Disassembly* above) and replace its battery by a fresh one. Dispose of old batteries responsibly in the appropriate in-store containers or at your local disposal center; do neither take them apart nor throw them in a bin nor in fire nor in your environment.[8]

Disposal of the calculator: Your *WP 43S* must not be disposed of along with household waste. Remove its battery (cf. *Inserting / replacing the battery* above) and dispose of your *WP 43S* according to applicable laws and regulations at your electronics collection point.

[8] No, we do not think you are stupid, irresponsible, and lack any experience. Blame the lawyers of the respective people and the courts of that great nation who made such a waste of print space inevitable.

Problem Solving, Part 1: First Steps

Start exploring your *WP 43S* by turning it on: Press its bottom right key – notice that **ON** is printed below that key. Doing this the very first time, you will get a display like this:



For turning your *WP 43S* off again, press the blue **g** (notice a little **g** appearing top left in the display), then press **EXIT** (which has **OFF** printed below it). Since your *WP 43S* features *Continuous Memory*, turning it off and on does not affect the information it contains (there is no “All Clear” at power up). For conserving battery power, your *WP 43S* will shut down automatically some ten minutes after you stopped using it – turn it on again and you can resume your work right where you left off.

This works as on preceding pocket calculators (like a *WP 34S* or an *HP-42S*). Your new *WP 43S*, however, looks cleaner than a *WP 34S* while more colorful than an *HP-42S.* This is due to your *WP 43S* featuring two *prefixes* **f** and **g** and *menus* – offering you up to four functions per calculator key.

Looking at an arbitrary one of its 41 black keys, white print is for the *primary* function of this key. For additional (*secondary*) functions, golden and blue labels are printed on the *key plate* above 37 keys, and grey characters are printed bottom right of 29 keys.

For accessing a primary function, just press the corresponding key. For a golden or blue (a.k.a. f- or g-*shifted*) function, first press the respective *prefix* f or g, then the corresponding key.

For better readability within the manuals, we will refer to keys using dark print on white from here on (like e.g. 1/x or EXIT). And referring to *secondary* functions (like x! or OFF), we will omit the *prefixes* f or g most times since redundant by color print.

Take the key ×, for **example**. Pressing...

- × alone will execute a multiplication,
- f + × will call x! calculating the *factorial* – e.g. press 9 x! and you will get $9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 362\,880$.
- g + × will call PROB, a *menu* of functions related to probability (EXIT will let you leave the *menu* again). Each label printed underlined on the keyplate refers to a *menu*.

- The grey letter **R** will become relevant when entering alphanumeric data.

Note all the 145 labels printed on the keyboard of your *WP 43S* are explained individually top left to bottom right in *App. 2* from p. 296 on.

Time for a little problem solving **example**:

Turn your *WP 43S* on again if necessary. Press ⬅. Your display will show 0. in each of its four rows then.

Now, let's assume you want to fence a rectangular patch of land, 40 *yards* long and 30 *yards* wide.[9] You have already set the 1st corner post (A), and also the 2nd (B) in a distance of 40 *yards* from A. Where do you set the 3rd and 4th corner posts (C and D) to be sure that the fence will form a proper rectangle? Simply key in:

3 0

```
                0.
                0.
                0.
30_
```

DISP FIX 0 1 [10]

```
               0.0
              30.0
```

Note the cursor vanished from the bottom row and the number 30 is adjusted to the right, indicating this input being closed now; so the next number can be entered:

4 0

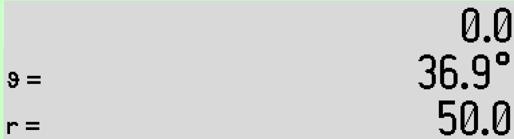
```
               0.0
              30.0
40_
```

---

[9] Assume this little patch of land in suburbia. We use outdated *British Imperial* units here so our US-American readers can follow, but this example will work with *meters* instead of *yards* as well.

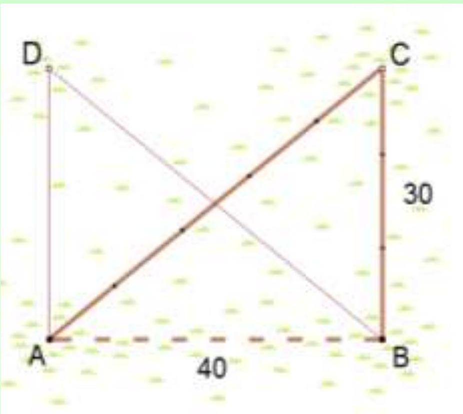
[10] Press g + E to access DISP; the *menu* DSP will open in the lower third of the screen. Press the leftmost top row key for FIX. Then enter 0 1 telling your *WP 43S* it shall display only 1 decimal digit (internally, everything is handled with full precision always). See *Section 2* for more about output formatting.

Generally, we will print no more than one display row containing just zero from here on for print space reasons.



All you need is the number in the bottom row,[11] a friend, and 80 *yards* of rope now:

Ask your friend to hold both ends of the rope firmly for you, take the loose loop and walk away as far as you can – when the loop is stretched, mark that position on the rope and return to your friend. Ask him or her to hold this point of the rope as well, fetch the two loose loops and walk again as far as possible – when the loops are stretched, mark both positions on the rope. Return again; hand over the two new points and walk once more, now with four loose loops. After marking as before, your rope will show marks every 10 *yards*.



Nail its one end on post A and its other end on B, fetch the loose loop and walk 5 marks away as calculated. As soon as both sections of the rope are tightly stretched, stop and place post C there. You may set post D the same way on the other side.

This method works for arbitrary rectangles, whatever other distances may apply (you will need a tapeline in the general case). As soon as you press →P, your *WP 43S* does the necessary calculation of the diagonal automatically for you. You just provide the land, posts, rope, hammer and nails. And it will be up to you to set the posts!

Another introductory **example** (basically quoted from the *HP-25 OH* though updated following progress in research in the meantime – only 12 moons of *Jupiter* were known in 1975): [12]

[11] Forget the number displayed above for the time being. We will talk about it later.

[12] Reading some of *HP*'s vintage calculator manuals may be fun and interesting for you. They are still available at low cost – together with almost complete information about

To calculate the surface area of a sphere, the formula $A = \pi\, d^2$ can be used, where $\boldsymbol{A}$ is the surface area, π is 3.141 5..., and $\boldsymbol{d}$ is the diameter of the sphere.

Ganymede, one of *Jupiter's* 79 moons, has a diameter of 5262 km. To use your *WP 43S* to manually compute the area of *Ganymede*, you can press the following keys in order:

5 2 6 2	5 262_	diameter of *Ganymede*
x^2	27 688 644	square of the diameter
π	3.1	the constant π (rounded to 1 decimal as set above)
×	86 986 440.6	area of *Ganymede* (in km^2, i.e. *square kilometers*).

If you wanted the surface areas of each of *Jupiter's* 79 moons, you could repeat the above procedure 79 times. However, you might wish to write a *program* that would calculate the area of a sphere from its diameter, instead of pressing all the keys for each moon.

To calculate the area of a sphere using a program, you should first write the program, then you must record the program into the calculator, and finally you run the program to calculate the answer.

Writing the program: You have already written it! A program is nothing more than the sequence of keystrokes you would execute to solve the same problem manually.

Recording the program: To record the keystrokes of the program into the calculator, press the following keys in order.

P/R	switch to *program-entry mode* (*PEM).*
GTO . .	go to the point in program memory where free space begins.

---

all the other *HP* calculators built between 1968 and 1990 – in one package on media distributed by the online *Museum of HP Calculators* (see *http://www.hpmuseum.org/cd/cddesc.htm*).

LBL L	This is the opening step of your program labelled **L** – for L just press +/- here as you see a grey L printed next to it.
x^2 π ×	These keys are the same you pressed to solve this problem manually above.
RTN	This is the closing step of the program. Finally,
EXIT	exits *PEM* and returns to *run mode*.

So a straight program on your *WP 43S* consists of an opening LBL step and a closing RTN framing the keystrokes you need for solving the respective problem manually.

Running the program: Now all you have to do to calculate the area of any sphere is keying in the value for its diameter and press

XEQ L (meaning 'execute program **L**').

When you press XEQ L the sequence of keystrokes you recorded is automatically executed by the calculator, giving you the same answer you would have obtained manually:

For example, to calculate the surface area of *Ganymede*, press

5 2 6 2	5 262_	*Ganymede's* diameter
XEQ L	86 986 440.6	its surface area – as you calculated manually above. So you know your routine works properly.

With the program you have recorded, you can now calculate the respective surface area of any of *Jupiter's* moons – in fact, of any sphere – using its diameter. You have only to leave the calculator in *run mode* and key in the diameter of each sphere that you wish to compute, and then press XEQ L. For example, to compute the surface area of *Jupiter's* moon *Io* with a diameter of 3643 km:

3 6 4 3	3 643_	*Io's* diameter
XEQ L	41 693 486.7	its surface area;

3 1 2 2	3 122_	*Europa's* diameter
XEQ L	30 620 739.2	its surface area;
4 8 2 1	4 821_	*Callisto's* diameter
XEQ L	73 017 025.3	its surface area; etc.

Programming your *WP 43S* is *that* easy! It remembers a series of keystrokes and then executes them automatically when you press XEQ ...[13]

There is no need memorizing a complicated formula after you keyed it in once – your *WP 43S* will remember it for you (and provides space for dozens more). Furthermore, you can even define individual shortcuts to your favorite routines by customizing the keyboard of your *WP 43S.*

The early portions of this handbook show you how easy it is to manually use the power of your *WP 43S*; while in *Section 3: Programming* you will find a complete guide to *WP 43S* calculator programming. Even if you have used other pocket calculators …, you will want to take a good look at this handbook. It explains the unique *HP* logic system that makes simple answers out of complex problems, and *WP 43S* features that make programming painless. When you see the simple power of your *WP 43S*, you'll become an apostle just as have some millions of *RPN* calculator owners before you.

First, let's demonstrate how to generally enter common decimal numbers in your *WP 43S*. Therefore, please return to *startup default* display format via DISP ALL 0 0.[14]

---

[13] Program **L** as recorded above is a very short one: its center part contains four keystrokes only (x^2 g π ×). You may store far, far more keystrokes in program memory – the overall procedure of storing and running programs, however, will remain unchanged. Only the center part of the program will grow.

Programming is comprehensively covered in *Section 3* of this manual.

[14] ALL may well be on your screen still since you called DISP on p. 19.

How to Enter Common Numbers (and How to Edit Them)

Numeric entry is as straightforward as typing: for **12.3456**, for instance, simply press 1 2 . 3 4 5 6 and you will see

12.345 6_ .

You may key in up to 43 digits at once, echoed immediately in the bottom numeric row of the screen (note the gap inserted automatically after each group of three digits for easier reading). Any digit mistyped may be erased by ← and can be replaced then.

For entering negative numbers such as **−7.8** , key in 7 +/- . 8 or 7 . +/- 8 or 7 . 8 +/- : pressing +/- changes the sign of the number being keyed in. Only negative signs will be displayed.

For a huge figure such as the age of the universe in years as we know it today, just enter 1 3 . 8 E 9 which is echoed

13.8×10^9_

in '*mantissa plus exponent*' format. The key E stands for '*enter exponent*'. Note your *WP 43S* allows for a naturally readable display instead of showing you cryptic machine formats like 13.8E 9 .

During numeric input, your keystrokes are generally just echoed in the bottom numeric row. Input is closed and released for interpretation by a command – e.g. by ENTER↑. Here, this will change the display to the equivalent:

13 800 000 000. .

Note the number moved to the right (cf. p. 20). Closed numbers in the bottom row may be cleared at once by pressing ←. This puts 0. in said row, and subsequent input will overwrite this 0. then.

Really tiny numbers such as a typical diameter of an atom (i.e. 0.000 000 000 1 m – with ten zeroes heading the digit 1) are entered in full analogy to huge numbers: E +/- 1 0 will do here, and this will be displayed when closed by EXIT as

0.000 000 000 1 in *startup default* format.

By the way, this may be shown significantly more compact as

$1.\times10^{-10}$ or even

$1,\cdot10^{-10}$ with other display settings (as treated in *Section 2*).

> Note you did not have to enter 1 E +/- 1 0 here: If there is no numeric input heading E, **1** is assumed for the mantissa per default. And pressing +/- after E will change the sign of the exponent – if you want to change the sign of the mantissa, press +/- before entering E or after closing the entire input.

There are also other numeric *data types* like integers, *times*, or *dates* available on your *WP 43S* – these (and more) will be covered in *Section 2* together with more output formats provided.

How to Enter and Execute Commands

This is easy as well: Just enter the keystrokes required to access the label calling the command you wish to be executed (cf. p. 19). Pending input will be echoed at left end of the top numeric row in the *LCD* until the command is completed. Therein, pending *prefixes* f or g will be echoed by f or g, if applicable; these characters will be replaced by the name of the command accessed as soon as it can be decoded.

For many commands, f or g will be the only echo you will really see during command entry since the next keystroke may well terminate command input already (as observed with g →P above), call and execute the command, and display the result.

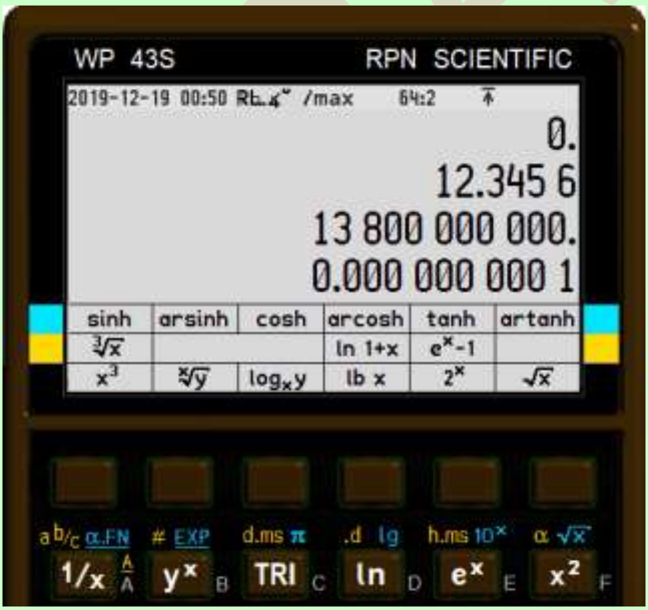
Some commands, however, require trailing parameters and will thus stay in the top numeric row for longer. STO and RCL are commands of this kind, and there are many more (see pp. 60f and the *ReM*).

Menus – Items à la carte

Your *WP 43S* features more than 650 operations, far too many for showing all of them on the keyboard. Hence most operations live in *menus*. In addition to commands, also arbitrary characters, constants, conversions, digits, programs, *submenus, system flags*, or variables defined may be stored in *menus*: we collectively call them *menu items* or simply *items*. By using *menus* we can keep the keyboard relatively tidy.

Your *WP 43S* features 30 *menus* on its keyboard, printed underlined for easy recognition there (except TRI).[15] In alphabetic order, these are ADV, BITS, CATALOG, CLK, CLR, CONST, CPX, DISP, EQN, EXP, FIN, FLAGS, INFO, INTS, I/O, LOOP, MATX, MODE, PARTS, PROB, P.FN, STAT, STK, TEST, TRI, U→, X.FN, α.FN, Σ, and ∠→.

Call any *menu* by simply accessing its label (cf. p. 19). This will open the *menu* and cause the lower part of the calculator screen (called the *menu section* from here on) displaying the respective *menu view*.



Example: Press EXP and EXP will open with a *menu view* as pictured here.

As long as this *view* is displayed, simply press, for example, ...

- the 2nd ☐ for $\sqrt[x]{y}$,
- f and the leftmost (1st) ☐ for $\sqrt[3]{x}$,
- g and the 3rd ☐ for cosh, known as the hyperbolic cosine.

[15] We print them underlined throughout this manual for the same reason. Note, however, they are stored in your *WP 43S* without that underline and hence displayed also in *menu views* without it. The labels A, <, and ● will be treated further below.

- If you would press [f] and the 2nd [ ], nothing will happen since no label is displayed there – no operation is linked to this [f]-shifted [ ].

We may as well print ∛x if we want to indicate the access path to this [f]-shifted [ ] in most compact way. In analogy, blue background may be printed for a [g]-shifted [ ] (like cosh here), and grey background for an unshifted [ ] function (like ˣ√y here).

Generally, whenever a *menu* is called, its top *view* will be displayed in the *menu section*. Any such *view* may contain up to 18 *items*:

- up to six assigned to the unshifted top row of keys,
- up to six to the [f]-shifted (note the golden stripes framing the *LCD* there), and
- up to six to the [g]-shifted top row of keys (note the blue stripes).

For calling a specific *item* contained in such a *view*, use the corresponding [ ], preceded by [f] or [g] if applicable (this is called a *softkey* from here on).

Any predefined *menu* may contain more than just one *view*. This will be indicated by a dashed line limiting the *menu section* on the screen. Whenever such a *multi-view menu* shows up, [▲] will advance to the next *view* and [▼] will return to the previous one, changing the labels displayed. Because *multi-view menus* are circular, also pressing [▲] repeatedly will return to the first *view* after all other stored *views* were displayed (thus, for a menu containing just two *views*, both [▲] and [▼] will display the next *menu view*).

> **Any *menu view* will stay constant** – granting direct access to the up to 18 functions displayed – **until you leave it** (e.g. via [▼] or [▲], if applicable, or via [EXIT]) **or call another *menu*.**

To indicate the access path via a *menu* and the corresponding *softkey*, we will generally print the background colors as explained at top of this page from here on. Note that *submenus* contained in a *menu* are displayed just inverted without the '*menu* underline'. Pressing [EXIT] in a *submenu* will bring you back to its parent *menu* (containing the label of said *submenu*).

How the Keyboard is Organized

You might have already recognized that labels on your *WP 43S* are printed grouped according to their purposes. Beyond the digits and the four arithmetic operations [+], [−], [×], and [/], five larger groups are provided:

Menu keys calling *items* from the *menu* displayed above

Modes, *data types*, and 'common' trans-cendental functions. SIN, COS, TAN, and their inverses are in TRI (no underline on this key for space reasons)

Stack and *register* operations

General navigation, information and con-trol keys: e.g. [⬅] for deleting, [↶] for un-doing the last com-mand, [▲] and [▼] for browsing, [EXIT] for general escaping

Functions for programming and calling programs: E.g. [XEQ] for executing a program, [R/S] for running or stopping it

[f], [g], and the *menus* in particular allow for easily accessing a multiple of the 43 primary functions this keyboard could take.

Before showing the operation of your *WP 43S* in detail, let's return to our introductory problem solving examples for four general remarks:

1. We presume you have graduated from an US *High School* at minimum, passed *Abitur*, *Matura*, or an equivalent graduation. So we will not explain basic mathematical rules and concepts here. Please turn to respective textbooks.
2. There is absolutely no need to enter units in your calculations: Just stay with a coherent set of units while calculating and you will get meaningful results within this set.[16] If you need to convert special inputs into *SI* units or require results expressed in particular units, however, U→ and L→ will help (see pp. 276ff).
3. Although you entered just integers for both edges of your little patch of land in the example on pp. 20f, your *WP 43S* calculated the diagonal using *real numbers*. This allows for decimals in input and output as well. Alternatively you may enter fractions such as e.g. 6 ¼ if this carries a benefit for you. Your *WP 43S* features also more *data types* – we will introduce them to you in *Section 2*.
4. In four decades of scientific pocket computing, a wealth of sample applications has been created and published by different authors – more and better than we can ever invent ourselves. We do not intend to copy all of them; instead, we recommend the media mentioned on p. 21 once again: they contain almost all the user guides, application handbooks, and manuals printed for vintage *HP* calculators in two heroic decades beginning with *HP*'s very first desktop calculator, the *HP-9100A* of 1968 (without any *IC*, but with a cathode ray tube built in). Be assured that all computations described there for any scientific or engineering calculator can be done on your *WP 43S* – most of them significantly faster and in a more elegant way. Nevertheless, we included more than 160 new and vintage examples in this manual to support you learning your new tool.

---

[16] A quick and simple unit analysis is strongly recommended before starting a calculation of a formula you may have derived yourself. The big advantage of *SI* is that this is the largest coherent set of units available on this planet. Unit prefixes in *SI* simply represent powers of 1000 (look up the *ReM* if necessary).

Problem Solving, Part 2: Elementary Stack Mechanics

Most of the commands your *WP 43S* provides are mathematical operations or functions taking and returning *real numbers*. *Real numbers* (or shortly *reals*) are numbers like 0.12 or 3.141 592 653 59 or -5.67×10^{-8}. Note that integers like 3 or 12 345 678 or −121, as well as fractions like 2/5 or 137/7 are mere subsets of *reals*.

Depending on the particular command you choose, it may operate on one, two, or three such numbers at once to generate a result. In spite of the over 650 functions available, you will find your *WP 43S* functions simple to operate by using a single, all-encompassing rule:

> **When you press a function key, your *WP 43S* will execute the operation assigned to it immediately** (if it requires parameters it will execute with parameter input completed).

One-number (*monadic*) functions: Many functions provided on your *WP 43S* operate on <u>one</u> number only.

Ten *monadic* functions are found on the keyboard, starting top left: the reciprocal [1/x], the logarithms [ln] and [lg], the exponentials [e^x] and [10^x], square [x^2] and square root [$\sqrt{x}$], [|x|] (making negative numbers positive), [+/-] (multiplying closed numbers by −1), and the factorial [x!].

Examples:

[8] [x^2]	returns	64	,
[1/x]		0.015 625	,
[+/-]		-0.015 625	,
[|x|]		0.015 625	,
[$\sqrt{x}$]		0.125	,
[1/x]		8.	,
[10^x]		100 000 000.	,
[lg]		8.	,
[e^x]		2 980.957 987 041 728	,
[ln]		8.	, and
[x!]	returns	40 320.	.

Generally, *monadic* functions replace the value (called x) displayed in the lowest numeric row on the screen before calling the function by the respective function result $f(x)$ (e.g. $f(x) = x!$ in the last example). Everything else on the screen stays as it was.[17]

Check the *IOI* for the many *monadic* functions provided (more logarithmic, exponential, root, trigonometric, and hyperbolic functions, unit conversions, etc.).

Two-number (*dyadic*) functions: Some of the most popular mathematical functions operate on <u>two</u> numbers and return one. Think of the four basic arithmetic operators + and −, × and / .

Example:

Assume owning an account of 1234 US$ and taking 56.7 US$ away from it. What will remain? An easy way for solving such a problem works as follows:

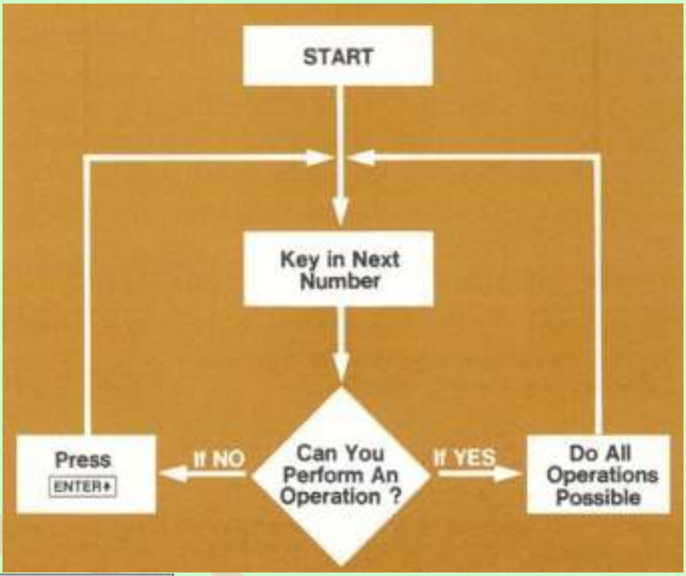
On a piece of paper		→	On your *WP 43S*		
Write down the 1st number:	1234		Key in the 1st number:	1 2 3 4	1 234_
Start a new line.			Close 1st input:	ENTER↑	
Write down the 2nd number:	56.7		Key in the 2nd number:	5 6 . 7	56.7_
Draw a line below.					
Subtract:	1177.3		Subtract:	−	1 177.3

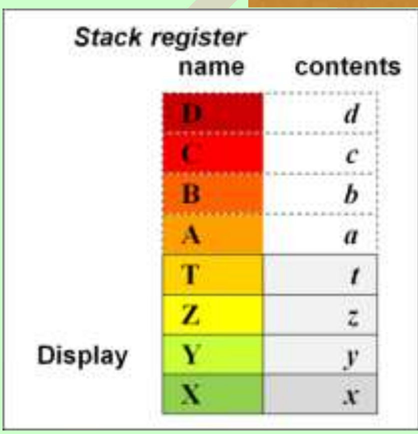
This is the essence of *RPN*:

Provide the necessary operands,
then execute the requested operation
by pressing the corresponding function key.

[17] Your *WP 43S* features also *monadic* functions operating on other data than *reals* and/or returning a different kind of data in output. These functions work the same way: x will be replaced by $f(x)$, everything else remains untouched.

HP itself explained this method using a very compact picture.[18] And a major advantage of *RPN* compared to all other calculator operating systems known to us is that it sticks to this basic rule – always.[19]





As the paper holds your operands while you are calculating manually, some space holding your operands on your *WP 43S* is required as well. The *stack* does this job. Think of it like a pile of *registers*, a work space for your calculations.

Bottom up, these *registers* are traditionally called **X**, **Y**, **Z**, and **T**,

---

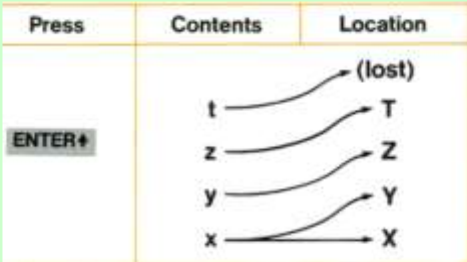
[18] This picture is copied from the brochure 'ENTER↑ *vs.* =' of 1974.

[19] This rule applies for functions regardless of the kind of objects they operate on. This way of writing operations is called *postfix* notation since the operator is entered after the operands (hence *RPN*, cf. footnote 1). It suits electronic calculating very well; and it eases work for human brains, too – see further below.

Some people might claim that the above global rule strictly holds for *RPL* only. *RPL* (meaning *Reverse Polish Lisp*) is a programming language and notation developed from *RPN* in the 1980's. Maybe those people are even right. In my opinion, however, *RPL* strains the *postfix* principle beyond the pain barrier, exceeding the limit where it becomes annoying for human brains. Not for everybody, of course, but also for many scientists and engineers. Thus, we stick to classic *RPN* on the *WP 43S* as we did on the *WP 34S* and *WP 31S.*

optionally followed by **A**, **B**, **C**, and **D** on your *WP 43S*.[20] New input is always entered in **X**, and at least ***x*** is always displayed in *run mode* – ***y, z,*** and ***t*** may be (so you may see the contents of up to four *stack registers* on the screen at the same time if you want).

ENTER↑ separates two input numbers by closing the first number ***x*** and copying it into **Y**, so **X** can take a new number then without loss of information (cf. above). The contents of the upper *stack registers* are lifted out of the way before. In a 4-*register stack*, ***z*** is copied into **T** and ***y*** into **Z** before ***x*** will be copied into **Y**.



This is the classical function of ENTER↑ from the *HP-35* of 1972 until the *HP-42S* ceased production in 1995. ENTER↑ affects all *stack registers*, and the previous content of the <u>top</u> *register* gets lost. It is often said ENTER↑ 'pushes ***x*** on the *stack*' (although it pushes ***x*** under the *stack* in the usual pictures).

Let's look at our account example again, putting it in a *stack diagram*:[21]

T				
Z				
Y		1 234	1 234	
X	1 234_	1 234	56.7_	1 177.3
Input	**1234**	ENTER↑	**56.7**	−

---

[20] This optional 8-*register stack* was invented by *Pauli* and me and launched with *WP 34S* in 2011. *WP 31S* features it as well. See the further text for its advantages.

[21] The *stack diagram* is presented here for a traditional 4-*register stack*. At beginning, some arbitrary data may be present in the upper *stack registers* **Y**, **Z**, and **T**, remaining from earlier operations. These data are irrelevant for this calculation, so we left them aside here; in further *stack* diagrams we will omit entirely all *stack registers* not containing any data relevant for the particular calculation, for sake of clarity and print space. And we will generally use plain bold text denoting numeric input from here on for the same reasons.

After having entered the 2nd number (**56.7**, the new x), pressing [−] subtracts this from the 1st number (**1234** in **Y**) and puts $f(x, y) = y - x =$ **1177.3** in **X**. This procedure applies to the overwhelming majority of functions featured on your *WP 43S*:

Put the operands on the *stack*,
then execute the operation $f(x, \ldots)$,
and the result will be displayed.[22]

A large part of mathematics is covered by such *dyadic* functions and combinations of them. Let's look at a chain calculation:

Example:

$$\frac{(12.3-45.6)\,(78.9+1.2)}{(3.4-5.6)^7}\ .$$

This is as a combination of six *dyadic* functions: two subtractions, one addition, a multiplication, an exponentiation, and a division.

And this is how that problem is solved on your *WP 43S*, starting top left in the formula, and what happens on the *stack* while solving:

Y		12.3	12.3	
X	12.3_	→ 12.3	45.6_	→ -33.3
Input	**12.3**	ENTER↑	**45.6**	[−]

You will have recognized that this 1st parenthesis in the numerator was solved exactly as demonstrated in our little account example above. Now proceed to the 2nd parenthesis:

Z		-33.3	-33.3		
Y	A -33.3	78.9	78.9	-33.3	
X	78.9_	78.9	1.2_	→ 80.1	-2 667.33
	78.9	ENTER↑	**1.2**	[+]	[×]

[22] This completely eliminates the need for an [=] on the keyboard.

It is solved like the first. Though in the 1st step of this sequence, the prior result (of 1st parenthesis) is lifted automatically (A) to **Y** to avoid overwriting it with the next number keyed in. This move is called *automatic stack lift*.

Actually, such an *automatic stack lift* works as if ENTER↑ was pressed before the first digit of the new input number (i.e. before **7** here). *Automatic stack lifting* is standard on *RPN* calculators, reducing the number of keystrokes necessary, and will not be indicated from here on anymore.[23] Remember you need pressing ENTER↑ just for separating two consecutive numbers in input – cf. the flow diagram on p. 33.

Due to *automatic stack lifting*, there is also no need for clearing your *WP 43S* before starting a new calculation – old data are just lifted out of the way when new input is entered. In consequence, we need neither any (nor any) and can solve problems with a minimum of keystrokes.

After having solved the 2nd parenthesis of the chain calculation by pressing +, the results of both upper parentheses were on the *stack* in **X** and **Y** – so everything was well prepared for the multiplication to complete the numerator. Thus, we just did it.

Now we start calculating the denominator – once again the intermediate result is lifted automatically in the 1st step:

	-2 667.33	-2 667.33		-2 667.33		
-2 667.33	3.4	3.4	-2 667.33	-2.2	-2 667.33	
3.4_	3.4	5.6_	-2.2	7_	-249.43…	10.693…
3.4	ENTER↑	**5.6**	−	**7**	y^x	/

Note the *automatic stack lift* when entering **7** affects two intermediate results now. Thus, everything is well prepared for the exponentiation in the

---

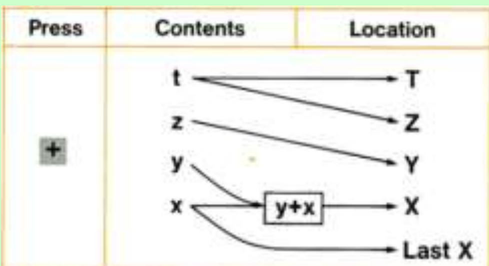
23 Also an *automatic stack lift* affects all *stack registers*, and the previous content of the top *register* gets lost again. Of all commands provided on your *WP 43S* (more than 650), there are only 4 disabling *automatic stack lift*: ENTER, CLX, Σ+, and Σ–. Some reasoning:

- After ENTER↑, you generally want to key in the consecutive number.
- CLX (called by CLX or ←) is for clearing **X** to make room for a corrected value instead of the one deleted (and we do not want a useless extra zero on the stack).
- Regarding Σ+ and Σ–, please see the chapters about statistical functions in *Section 2* for the reasons.

penultimate step and the final division of the numerator (in **Y**) by the denominator (in **X**). Voilà!

Following this example, you have seen the five most popular *dyadic* functions in action: [+], [−], [×], [/], and [y^x]. Your *WP 43S* provides many more *dyadic* functions.

As you have observed several times now, the contents of the *stack registers* drop whenever a *dyadic* function is executed. Like the *automatic stack lift* mentioned above, also this *automatic stack drop* affects all *stack registers* as pictured here:



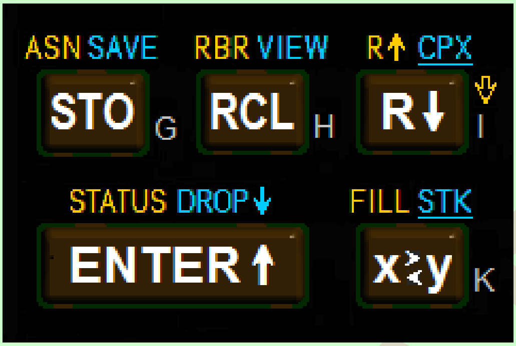
x and ***y*** are combined resulting in $f(x, y) = y + x$ put into **X**; then ***z*** drops to **Y**, and ***t*** to **Z**; since nothing is available above ***t*** on a 4-*register stack* for dropping, the top *register* content is repeated (see also p. 40 – 'Last X' will be covered on p. 50).

There are also a few **three-number (*triadic*) functions** featured (e.g. →DATE, %MRR). Executing such a function replaces ***x*** by $f(x, y, z)$; then ***t*** drops into **Y** and so on, and the content of the top *stack register* is repeated twice (see p. 40 for an example). All *triadic* functions provided on your *WP 43S* are found in *menus*.

And some functions operate on one number but return two (like DECOMP) or three (e.g. DATE→). Other operations do not consume any *stack* input at all but may just return one, two, or three objects (like RCL, SUM, or L.R.). Then these extra objects will be pushed on the *stack*, taking one *register* each (see p. 40).

Looking Closer at the Automatic Stack

For understanding the genius of *RPN*, we will look a bit closer to the functions operating on the *stack*. In addition to the one-, two-, and three-number (*monadic*, *dyadic*, and *triadic*) functions explained in previous chapter, there are some dedicated *stack* and *register* commands:



The memory control operations ENTER↑, x≷y, R↓, R↑, STO, RCL, and VIEW are known from previous *RPN* calculators already. They are all found within this small area of the keyboard, together with RBR, FILL, DROP↓, and STK.

Your *WP 43S* contains even more *stack* and *register* commands, e.g. CLSTK, CLREGS, DROPy, STOCFG and RCLCFG, STOS and RCLS, x≷, y≷, z≷, t≷, and ≷. Most of them are found in STK.

And your *WP 43S* allows for expanding the traditional 4-*register stack* to eight *registers*: just enter

FLAGS SF SYS.FL SSIZE8.

In consequence, the fate of *stack* contents will depend on the particular operation executed as well as on the *stack* size set at execution time. Operations on the 4-*register stack* work as known from vintage *HP RPN* calculators since the *HP-45*. On the optional 8-*register stack* of your *WP 43S*, everything works in analogy – just with more *registers* available for intermediate results; so you will hardly ever run into a *stack overflow* (see p. 44 for an example).

Please find below what ENTER↑, FILL, DROP↓, DROPy, x≷y, R↓, R↑, and further representative functions do in detail on *stacks* of either size. Then you will also know why ENTER↑ and the *stack* rotation command R↑ show arrows pointing up while R↓ and DROP↓ point down.

			Stack contents after executing …						
	Stack register	Assumed initial contents	ENTER↑	FILL	DROP↓	DROP y	x≷y	R↓	R↑
With **4** *stack registers*	**T**	***t*** = 4.	3.	1.1	4.	4.	4.	1.1	3.
	Z	***z*** = 3.	2.	1.1	4.	4.	3.	4.	2.
	Y	***y*** = 2.	1.1	1.1	3.	3.	1.1	3.	1.1
	X	***x*** = 1.1	1.1	1.1	2.	1.1	2.	2.	4.
With **8** *stack registers*	**D**	***d*** = 8.	7.	1.1	8.	8.	8.	1.1	7.
	C	***c*** = 7.	6.	1.1	8.	8.	7.	8.	6.
	B	***b*** = 6.	5.	1.1	7.	7.	6.	7.	5.
	A	***a*** = 5.	4.	1.1	6.	6.	5.	6.	4.
	T	***t*** = 4.	3.	1.1	5.	5.	4.	5.	3.
	Z	***z*** = 3.	2.	1.1	4.	4.	3.	4.	2.
	Y	***y*** = 2.	1.1	1.1	3.	3.	1.1	3.	1.1
	X	***x*** = 1.1	1.1	1.1	2.	1.1	2.	2.	8.

Clearing the entire *stack* can be done by 0 FILL most easily. Nevertheless, a dedicated command CLSTK is provided in CLR for backward compatibility and program space economy (see p. 52).

x≷y takes the initial *stack* contents (as listed in the third column left) and swaps the contents of *registers* **X** and **Y**. Depending on the problems you solve and the way you proceed, you may sometimes find that *x* and *y* should be swapped before executing e.g. −, /, or y^x.

R↓ and R↑ may come handy for reviewing *stack registers* else unseen (unless you use the register browser RBR – see *Section 5*).

Stack register	Assumed initial contents		*Stack* contents after executing …					
			RCL L [24]	s [25]	x^2 [26]	+ [27]	→DATE [28]	DATE→ [29]
T	t = 4.		3.	2.	4.	4.	4.	2.
Z	z = 3.		2.	1.1	3.	4.	4.	20.
Y	y = 2.		1.1	s_y	2.	3.	4.	10.
X	x = 1.1		*last x*	s_x	1.21	3.1	1-02-03	1.
D	d = 8.		7.	6.	8.	8.	8.	6.
C	c = 7.		6.	5.	7.	8.	8.	5.
B	b = 6.		5.	4.	6.	7.	8.	4.
A	a = 5.		4.	3.	5.	6.	7.	3.
T	t = 4.		3.	2.	4.	5.	6.	2.
Z	z = 3.		2.	1.1	3.	4.	5.	20
Y	y = 2.		1.1	s_y	2.	3.	4.	10
X	x = 1.1		*last x*	s_x	1.21	3.1	1-02-03	1

RCL L represents the vintage command LASTx (see p. 50 for more about it). Note that the previous contents of the top *stack register* are lost when ENTER↑ or RCL are executed. Functions like s or DATE→

---

[24] This represents an arbitrary function pushing one object on the *stack*.

[25] This represents an arbitrary function pushing two objects on the *stack*.

[26] This represents an arbitrary *monadic* function.

[27] This represents an arbitrary *dyadic* function.

[28] Assume →DATE is called in *startup default* date mode (i.e. YMD). It represents an arbitrary *triadic* function here.

[29] Assume **1.102** or **1-10-20** in **X** initially here and *startup default mode* set, cf. *Sect. 2*.

will even cost the contents of two *stack registers*,. We recommend mitigating the effects of such losses by setting the *stack* to eight *registers* (cf. p. 38). – Please see the *IOI* for further information about the commands mentioned above.

Problem Solving, Part 3: The Stack in Advanced Calculations

Using the *stack* as described, *RPN* eliminates the need for an [=] key as well as for any parentheses [(] [)] keys. See the following **example**, showing a more elaborate formula than above. Find below a way for solving it step by step, and the corresponding *stack* diagrams. Enter [MODE] RAD and start calculating at the red 7:

$$2+\sqrt{\frac{1+\left|\left(\frac{30}{7}-7.6\times 0.8\right)^4-\left(\sqrt{5.1}-\frac{6}{5}\right)^2\right|^{0.3}}{\left\{sin\left[\pi\left(\frac{7}{4}-\frac{5}{6}\right)\right]+1.7(6.5+5.9)^{3/7}\right\}^2-3.5}}$$

Z						1.75	1.75		
Y		7	7		1.75	5	5	1.75	
X	7_	7	4_	1.75	5_	5	6_	0.83...	0.91...
	7	ENT↑	4	/	5	ENT↑	6	/	−

								0.25...	0.25...
				0.25...	0.25...		0.25...	12.4	12.4
0.91...			0.25...	6.5	6.5	0.25...	12.4	3	3
3.14...	2.87...	0.25...	6.5_	6.5	5.9_	12.4	3_	3	7_
π	×	TRI sin	6.5	ENT↑	5.9	+	3	ENT↑	7

0.25...		0.25...					
12.4	0.25...	2.94...	0.25...			27.6...	
0.42...	2.94...	1.7_	5.00...	5.25...	27.6...	3.5_	24.1...
/	y^x	1.7	×	+	x^2	3.5	−

This was the solution of the entire denominator. Let's continue with calculating the numerator now, basically following the same procedure, i.e. calculating from inside out (as you would do with pencil and paper):

					24.1...	24.1...			
	24.1...	24.1...		24.1...	6.08	6.08	24.1...		24.1...
24.1...	7.6	7.6	24.1...	6.08	30	30	6.08	24.1...	1.79...
7.6_	7.6	.8_	6.08	30_	30	7_	4.28...	1.79...	4_
7.6	ENT↑	**.8**	×	**30**	ENT↑	**7**	/	−	**4**

		24.1...	24.1...		24.1...	24.1...			
	24.1...	10.3...	10.3...	24.1...	10.3...	10.3...	24.1...	24.1...	
24.1...	10.3...	6	6	10.3...	1.2	1.2	10.3...	10.3...	24.1...
10.3...	6_	6	5_	1.2	5.1_	2.25...	-1.05...	1.12...	9.24...
y^x	**6**	ENT↑	**5**	/	**5.1**	$\sqrt{x}$	−	x^2	−

	24.1...		24.1...						
24.1...	9.24...	24.1...	1.94...	24.1...	2.94...			0.34...	
9.24...	.3_	1.94...	1_	2.94...	24.1...	0.12...	0.34...	2_	2.349...
|x| [30]	**.3**	y^x	**1**	+	x⇄y	/	$\sqrt{x}$	**2**	+

Even solving this formula requires only four *stack registers*.[31] Note there are no pending operations – each operation is executed individually, one

[30] You would not execute this step manually since you will see immediately that ***x*** is positive. In an automatic evaluation of such a formula, however, this step is important unless you know in advance that a negative intermediate result will not occur.

[31] We admit we were cautious seeing this formula and set SSIZE8 before starting the calculation here.

Additional remark: In the fifth step of last diagram, we have got the complete result for the numerator in **X**. And the result for the denominator is in **Y** whereto it silently traveled during all the other calculations (see above). In the subsequent step, we swapped ***x*** and ***y*** to put the operands in proper order for division of the numerator ***y*** by the denominator ***x***.

at a time, allowing **perfect control of each and every intermediate result.**[32]

> Note this is another characteristic advantage of *RPN*. In many real-life applications, intermediate results carry their own value, so further calculations may depend on the numbers you see there – this is called '*exploratory math*' and may well occur more frequently in your professional work than evaluating textbook formulas.

Experienced *RPN* calculator users have determined that by **starting every problem at its innermost number or parenthesis** and working outwards, you maximize the efficiency and power of your calculator.

> If, instead, you had tried solving the formula on p. 41 starting with the numerator of the root straight ahead, stubbornly calculating from left to right, you would have needed more keystrokes and six *stack registers* for the entire solution instead of only four (the colors in the record below represent the top *stack register* involved in each step):
>
> 1 [ENTER↑] 30 [ENTER↑] 7 [/] 7.6 [ENTER↑] .8 [×] [−] 4 [y^x]
> 5.1 [$\sqrt{x}$] 6 [ENTER↑] 5 [/] [−] [x^2] [−] [|x|] .3 [y^x] [+]
> [RAD] [π] [ENTER↑] 7 [ENTER↑] 4 [/] 5 [ENTER↑] 6 [/] [−] [×] [sin]
> 1.7 [ENTER↑] 6.5 [ENTER↑] 5.9 [+] 3 [ENTER↑] 7 [/] [y^x] [×]
> [+] [x^2] 3.5 [−] [/] [$\sqrt{x}$] 2 [+]
>
> Admittedly, this way is not very smart though you see it is viable.

There are, however, some problems where four *stack registers* will just not suffice <u>regardless</u> of the way you tackle with them:

> **Example:**
>
> Solve $\dfrac{(1+2)(9+8)+(3+4)(11+6)}{(5-7)(10+12)-(13+14)(15+16)}$.
>
> This highly symmetric formula lacks an unambiguous 'inside', so it does not matter where we start solving it. Let's begin with the numerator:

[32] Thus, operator precedence is your job. Look up *App. 1* for confirmation or reminder.

Z						3	3			
Y		1	1		3	9	9	3		51
X	1_	1	2_	3	9_	9	8_	17	51	3_
	1	ENT↑	**2**	+	**9**	ENT↑	**8**	+	×	**3**

T					51	51				
Z	51	51		51	7	7	51			
Y	3	3	51	7	11	11	7	51		170
X	3	4_	7	11_	11	6_	17	119	170	5_
	ENT↑	**4**	+	**11**	ENT↑	**6**	+	×	+	**5**

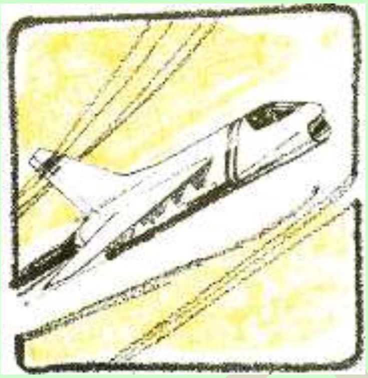
T					170	170				170
Z	170	170		170	-2	-2	170		170	-44
Y	5	5	170	-2	10	10	-2	170	-44	13
X	5	7_	-2	10_	10	12_	22	-44	13_	13
	ENT↑	**7**	−	**10**	ENT↑	**12**	+	×	**13**	ENT↑

A				170	170				
T	170		170	-44	-44	170			
Z	-44	170	-44	27	27	-44	170		
Y	13	-44	27	15	15	27	-44	170	
X	14	27	15_	15	16_	31	837	-881	-0.192 96…
	14	+	**15**	ENT↑	**16**	+	×	−	/

If you had set your *WP 43S* to four *stack registers* (as all of *HP's* pocket calculators featured so far), however, the last *stack* diagram would have deviated since *register* **A** could not be loaded automatically then:

T	170	170	170	-44	-44	-44	-44	-44	-44
Z	-44	170	-44	27	27	-44	-44	-44	-44
Y	13	-44	27	15	15	27	-44	-44	-44
X	14	27	15_	15	16_	31	837	-881	0.049 94…
	14	+	**15**	ENT↑	**16**	+	×	−	/

Then it would return a wrong result due to *stack overflow* in step 4 and subsequent repetition of wrong top *register* contents. Note this is possible – and there is (and will be) no warning since your *WP 43S* cannot know what you still need or what may be discarded without a problem.[33] Thus, we recommend setting SSIZE8 to play safe.



We will close this chapter with another real-life **example**:

For decades, solving the following formula for the *Mach number* of an airplane as a function of its *calibrated airspeed* (*CAS*) in *knots*[34] (here: 350) and *pressure altitude* (*PA*) in *feet* (here: 25 500) was used for demonstrating the simplicity and coherence of *RPN*:

$$\sqrt{5\left(\left[\left\{\left(1+0.2\left[\frac{CAS}{661.5}\right]^2\right)^{3.5}-1\right\}\{1-6.875\times10^{-6}\times PA\}^{-5.2656}+1\right]^{0.286}-1\right)}$$

Solve it like this:

350 [ENTER↑] **661.5** [/] [x^2] **.2** [×] **1** [+] **3.5** [y^x] **1** [−]
6.875 [E] [+/-] **6** [ENTER↑] **25500** [×] [+/-] **1** [+] **5.2656** [+/-] [y^x] [×]
1 [+] **.286** [y^x] **1** [−]
5 [×] [$\sqrt{x}$] resulting in 0.84, i.e. 84% of the speed of sound. You need only three *stack registers* for solving this.

---

[33] Assuming you begin your calculations with a clear *stack*, you could think of writing a little routine checking the contents of the top *stack register*, and displaying a warning when this *register* deviates from zero. Though that routine will turn useless at this very moment since the top *stack register* contents will stay nonzero further on. See previous page and trick #1 three pages below.

[34] The ancient *British Imperial* unit *knot* survived in aviation business and navigation:
$1\ knot = 1\ {}^{nautical\ mile}/_{hour} = {}^{463}/_{900}\ {}^{m}/_{s} \approx 0.5144\ {}^{m}/_{s} \approx 1.85\ {}^{km}/_{h}.$

The *foot* is another one from that heap of pre-modern units made obsolete by *SI* for two centuries (see U→ in *Sect. 5*). It also survived in aviation. We quote that US-American *Mach*-number formula without having verified it.

Translator's note for Europeans: *CAS* does not mean *C·A·S*, and *PA* does not mean *P·A* in that formula!

As you have seen, the way to solve a problem using *RPN* stays the same regardless of the problem size. You are always in control.

With an 8-*register stack* as provided on your *WP 43S*, you will be on the safe side, even dealing with the most advanced mathematical expressions you will meet in your professional life as a scientist or engineer. Promised.[35]

Let's quote a part of the *HP-25 OH* once more, just replacing all strings '*HP-25*' by '*WP 43S*':

> Now that you've learned how to use the calculator, you can begin to fully appreciate the benefits of the *Hewlett-Packard* logic system. With this system, you enter numbers using a parenthesis-free, unambiguous method called *RPN* (*Reverse Polish Notation*).
>
> It is this unique system that gives you all these calculating advantages whether you're writing keystrokes for a *WP 43S* program or using the *WP 43S* under manual control:
>
> - *You never have to work with more than one function at a time.* The *WP 43S* cuts problems down to size instead of making them more complex.
> - *Pressing a function key immediately executes the function.* You work naturally through complicated problems, with fewer key-strokes and less time spent.
> - *Intermediate results appear as they are calculated.* There are no "hidden" calculations, and you can check each step as you go.
> - *Intermediate results are automatically handled.* You don't have to write down long intermediate answers when you work a problem.
> - *Intermediate answers are automatically inserted into the problem on a last-in, first-out basis.* You don't have to remember where they are and then summon them.

[35] Of course, constructing an example leading to *stack* overflow even for eight *registers* is trivial. But first of all it will be exactly that: a constructed example – no real-world formula. And last not least, we assume there will be still an intelligent person operating the calculator, solving from inside out as recommended above.

- *You can calculate in the same order you do with pencil and paper.* You don't have to think the problem through ahead of time.

RPN takes a few minutes to learn. But you'll be amply rewarded by the ease with which the *WP 43S* solves the longest, most complex equations. With *RPN*, the investment of a few moments of learning yields a lifetime of mathematical bliss.

And calculations with other *data types* (see *Section 2*) follow the same simple rules. So at the bottom line, we recommend:

Set SSIZE8 and
let your *WP 43S* care for the arithmetic
while you care for the mathematics![36]

---

[36] You might ask: With the opportunity of an *8-register stack*, why are there only up to four *stack registers* displayed, not more?

The reason is simple: Once you have accustomed to *RPN*, you know the way it deals with your data on the *stack*. Consistently. Thus, watching the entire *stack* mechanics reliably working all the time does not carry any valuable information and will become boring or even distracting very soon.

Actually, the overwhelming majority of *RPN* pocket calculators displayed x only although there were $\mathbf{Y}$, $\mathbf{Z}$, and $\mathbf{T}$ quietly acting unseen always. Users were doing all sorts of tricks on that *stack* – just tracking y, z, and t in their minds. Even *HP's RPL* calculators (although they feature a so-called 'infinite' *stack*) did and do not display more than four *registers*.

Assuming people's mental abilities did not deteriorate generally in last decades, displaying more than four *stack registers* carries no lasting benefit. This holds especially since the odds for *stack overflow* in real-world calculations are reduced to zero when you follow our recommendations above. For the same reason, we omit heading indicators X, Y, Z, and T in display. Since you chose this calculator for yourself, you are obviously able to remember these four letters naming the bottom four *stack registers*.

On the other hand, if you feel distracted or even annoyed by the screen showing more than necessary, you may reduce the number of *stack registers* displayed to three, two, or even just one (using DSTACK), letting your brains compete with the ones of your fellow *RPN* users since 1972. Free space will flow in the display top down – x will always be displayed directly above the menu section. And multi-line output will be shown entirely always, regardless of current DSTACK setting.

We count on your abilities and are very confident you will succeed.

Special Tricks, #1: Top Stack Level Repetition

Whenever a *dyadic* or *triadic* function is executed, the *stack* will drop and the content of its top *register* will be repeated as illustrated on pp. 37 and 40. You may employ this *top stack register repetition* for some nice tricks.

See the following compound interest calculation: [37]

Example:

Assume your bank pays you 3.25% p.a.[38] on an amount of 15 000 US$; what would be your status after 2, 3, 5, and 8 years?

Solution:

Here, you are interested in currency values only, so set the display format by DISP FIX 2 . This causes the output being rounded to cents (internally, numbers are kept and calculated with far higher precision):

T		1.03	1.03	1.03	1.03	1.03	1.03
Z		1.03	1.03	1.03	1.03	1.03	1.03
Y		1.03	1.03	1.03	1.03	1.03	1.03
X	1.032 5_	1.03	15 000_	15 990.84	16 510.55	17 601.17	19 373.66
	1.0325	FILL	**15000**	× ×	×	× ×	× × ×
			after	2 years	3 years	5 years	8 years

Each multiplication consumes x and y for the new product $x \times y$ put in $\mathbf{X}$, followed by z dropping into $\mathbf{Y}$, and t copied into $\mathbf{Z}$. Due to *top stack register repetition* the interest rate is automatically kept as a constant on the *stack*, so the accumulated capital value computation becomes a simple series of × strokes.

This is demonstrated here for a 4-*register stack*. It works for an 8-*register stack* as well – with the contents of **D** repeated then.

---

[37] Translator's note for German readers: *Compound interest* = Zinseszins.

[38] Those were the times, my friend... ! Inflation was balancing those interest rates but saving was definitely more fun then, nevertheless.

Debt calculations are significantly more complicated – so avoid debts whenever possible! In the long run, it is better for you and your economy. Nevertheless, you can cope with such calculations as well using your *WP 43S* (see *Section 5*).

Another application making use of top *stack register* repetition is the *Horner scheme* for calculating polynomials. It tells:

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$
$$= (\ldots(a_n x + a_{n-1})x + \cdots + a_1)x + a_0$$

Example:

Solve $7 + 6.4x - 2.1x^2 + 5.2x^3 - 3x^4$ for $x = 0.908$.

Solution:

This problem can be rewritten to

$$\{\,[\,(-3x + 5.2\,)\,x - 2.1\,]\,x + 6.4\,\}\,x + 7$$

and is easily solved this way (with the display set to DISP FIX 0 1):

	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9
	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9
	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9
.908_	0.9	-3_	-2.7	5.2_	2.3	2.1_	0.1	6.4_	5.9	12.9
.908	FILL	**3** +/-	×	**5.2**	+ ×	**2.1**	− ×	**6.4**	+ ×	**7** +

Note how the ***x*** values float automatically down the stack to be used in multiplications.

FILL loads the entire *stack* always – be it 4 or 8 *registers* deep – it is far more convenient than hitting ENTER↑ multiple times.

Special Tricks, #2: LASTx for Reusing Numbers

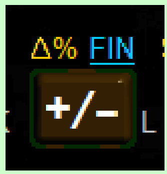
Your *WP 43S* copies ***x*** into the special *register* **L** (for 'Last ***x***') automatically just before a function is executed – as previous *RPN* calculators did (cf. the picture on p. 37). What is the benefit for you?

Example (from the *HP-15C OH*):

Two close stellar neighbors of Earth are *Rigel Centaurus* [39] (4.3 *light-years* away) and *Sirius* (8.7 *light-years* away). Use the speed of light, c ($2.997\,92 \times 10^8$ *meters/second*, or $9.460\,54 \times 10^{15}$ *meters/year*), to figure the distances to these stars in *meters*.

Solution (with SCI 1 set):

4.3	4.3_
ENTER↑	4.3
9.46073 E 15	$9.460\,54\times10^{15}$_
×	4.1×10^{16}
8.7	8.7_
RCL L	9.5×10^{15}
×	8.2×10^{16}



RCL L is reached by pressing **RCL**, then **+/-**; note the grey L printed bottom right of **+/-**.

Result: *Rigel Centaurus* has a distance of 4.1×10^{16} m (or 4.1×10^{13} km) to our planet, *Sirius* 8.2×10^{13} km.

So, recalling the last ***x*** via **RCL L** may save you from keying in lengthy numbers more than once. It also allows for reusing intermediate results without the need for storing them explicitly.[40]

---

[39] This is identical with *Alpha Centauri*. *Rigel* usually means a star in constellation *Orion*.

[40] There are only very few commands changing ***x*** but not loading **L**. Those are mentioned explicitly in the *IOI*. – Allocating a dedicated label to LASTx on the keyboard (like on the *HP-42S*) would not pay here since no keystrokes will be saved.

Error Recovery: [←] , [EXIT], and [↶]

Nobody is perfect – errors will happen although you are equipped with such a powerful tool. Stay cool – your *WP 43S* allows you undoing the last command executed, restoring the calculator state exactly as it was before that error occurred.

1. If you receive an **error message** in response to your function call, press [←] or [EXIT] ; this will erase that message and return to the state before that error happened (see pp. 68 and 308f). Then do it right!

2. If you have erroneously executed a **wrong function**, just press [f] [↶] to undo it immediately. [↶] recalls the entire calculator state as it was before that wrong operation was executed. Then resume calculating where you were interrupted. [41]

Example:

Assume – while you were watching an attractive fellow student or collaborator – you pressed [×] inadvertently instead of [/] in the fourth last step solving the lengthy formula on p. 41. Murphy's Law! Luckily, however, there is absolutely no need to start that calculation all over again – that error is easily undone as follows:

T	yzx	…	yzx	…	…
Z	xyz	yzx	xyz	yzx	yzx
Y	*numerator*	xyz	*num*	xyz	xyz
X	*denominator*	*num × den*	*den*	*num / den*	*correct result*
		[×]	[↶]	[/]	[√x]
	Fine so far.	**Oops!**	Undo	Resume	

So don't worry – be happy!

---

[41] [↶] operates on *stack*, statistic registers and *system flags*. Note, however, [↶] will not revert any operations you have confirmed explicitly (like RESET, see next page). And [↶] will undo the very last operation before [↶] only, nothing more – i.e. [↶] [↶] = [↶].
Previous *RPN* calculators used LASTx for error correction – [↶] works easier and more comprehensive.

Clearing and Resetting Your WP 43S

There are several ways you can remove obsolete information from your *WP 43S*. The most basic one is ⬅ – you have learned about it on p. 25. Almost all other clearing commands are contained in CLR:

CLX	Clears *stack register* **X** (i.e. sets it to zero)	CLSTK	Clears all *stack registers*
CLΣ	Clears all statistical *registers*	CLREGS	Clears all global and local *GP registers*[42]
CF	Clears the *flag* specified	CLFALL	Clears all *user flags*
CLP	Clears *current program*	CLPALL	Clears all programs
CLMENU	Clears the *programmable menu*	CLCVAR	Clears all variables of the *current program*
CLALL	Clears all programs and data (variables, *user flags*, and all *registers* including the *stack*)[43]	RESET	Resets your *WP 43S* to *startup default* (just *flash memory* contents will stay untouched)[44]

For your reference, *startup default* settings are:

> 2COMPL, ALL **0**, DEG, DENMAX **9999**, DSTACK **4**, GAP **3**, J/G **1752-09-14**, LinF, LocR **0**, RM **0**, TDISP **−1**, WSIZE **64**, and Y.MD. RANGE is set to 6145.
>
> The *system flags* AUTOFF, DECIM**.**, DENANY, MULT×, TDM24, and αCAP are set, all others are clear.

Red commands ask you for confirmation. Turn to the *ReM* for more information about the commands and *system flags* mentioned above.

[42] Find more about *GP registers* in next chapter. Note *stack* and statistical *registers* as well as variables are not touched by CLREGS.

[43] Note display formats as well as other user settings and assignments will remain unchanged. Only RESET clears everything except *flash memory* (see *Sect. 3* and *6*).

[44] If you cannot reach RESET for any reason whatsoever, a hard reset will do almost the same. Use the RESET hole on the calculator back side.

That's almost all you have to know about number crunching for the time being – calling commands and calculations with *real numbers* on the *stack*. Such capabilities did suffice for high flying applications already – see the picture above. There are, however, far more places than just the *stack* where you may store and save your data in your *WP 43S*. Let's present them to you.

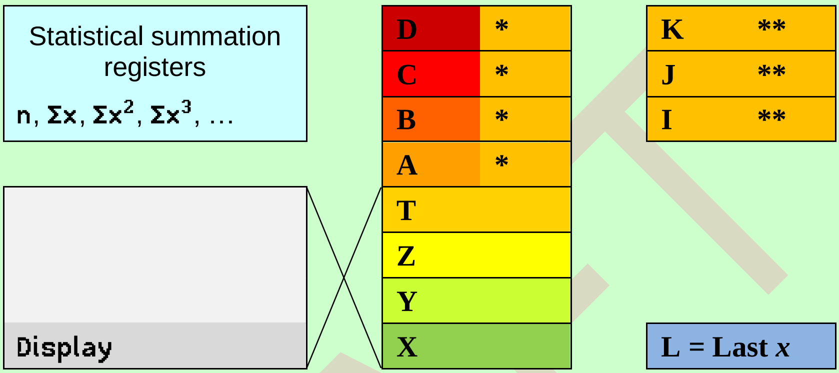
Addressing and Manipulating Objects in RAM

You have learned about the *stack* providing work space and temporary storage during your calculations. For long term storage, feel free to use other *registers*, variables, *flags*, and program memory. The remaining chapters of this section will tell you how to use the first three.

The pictures on the next two pages show the entire address space of your *WP 43S*. Depending on the way you configure its memory, a subset of all these addresses will be accessible for you.

Depending on the *stack* size you choose, either **T** or **D** will be the top *stack register*; **A – D** will be allocated for the 8-*register stack* if applicable. **I, J,** and **K** may carry parameters of statistical distributions (see pp. 97ff); **I** and **J** will also serve as *index pointers* in matrix editing (see pp. 163ff), and **K** is also the default *alpha register* for some special operations (see pp. 230f). Unless required for the purposes mentioned, **A**, **B**, **C**, **D**, **I**, **J**, and **K** may be employed as additional global *GP registers*.

Special *registers* and *stack*

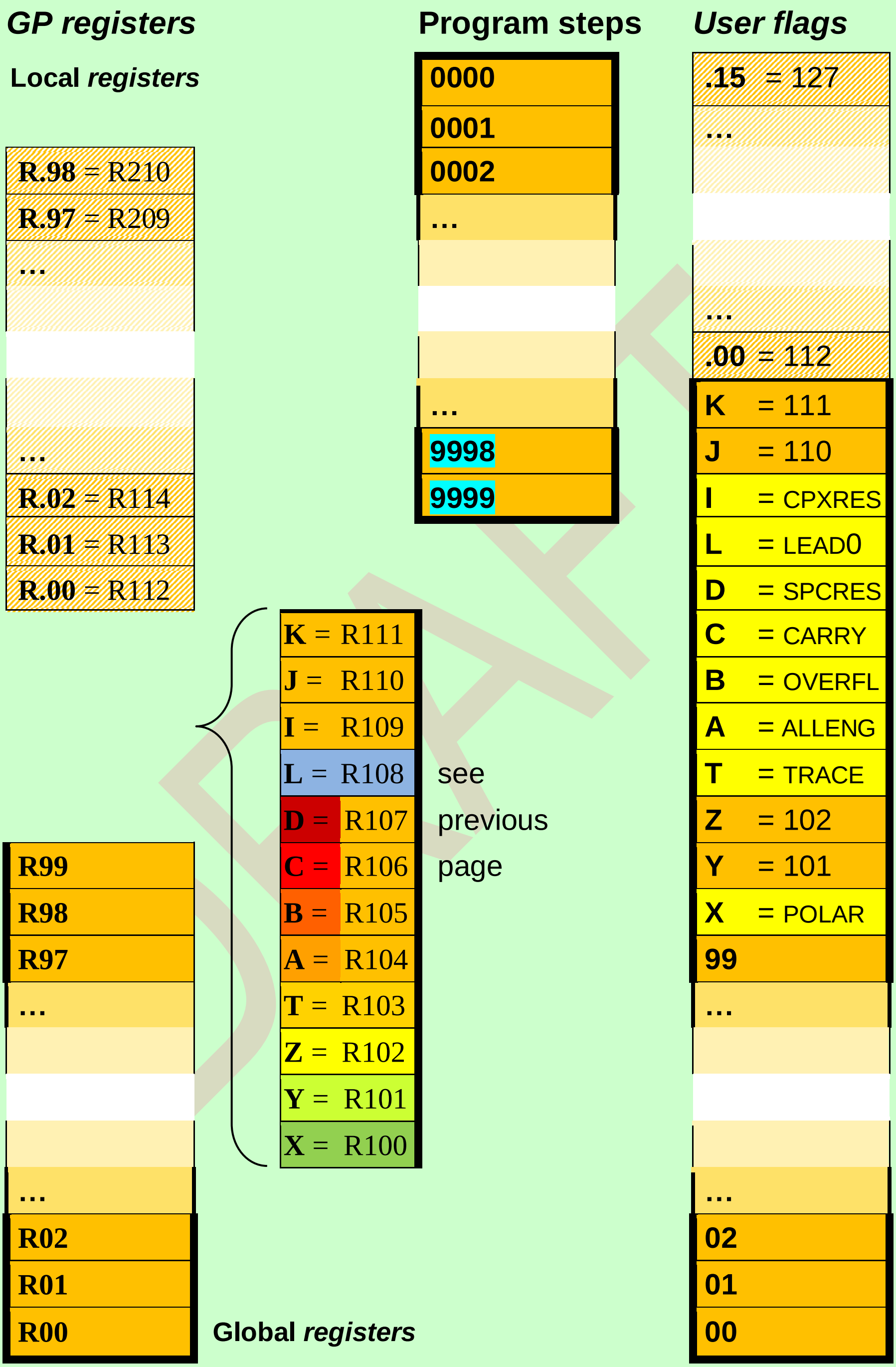


Turn overleaf to see all *registers* available as well as all *user flags*. Generally, *registers* or *flags* can be addressed as shown in the tables on pp. 60ff. Addresses ≥112 are used for local data (see pp. 233f).

Flags are elementary *items* having only two states, *set* and *clear*. You may think of them as switches being either on or off. You can employ *user flags* for signaling whatever you want. There are also *system flags* reflecting specific system states (overlapping with some lettered *user flags* for easier access, see also the *ReM*). Since *flags* are most useful in programming, they will be dealt with in *Section 3*.

Statistical data are accumulated in a set of dedicated summation *registers* not interfering with your other data (like in *WP 34S* and *31S* before). You may enter your gathered statistical data value by value, point by point, or in a single matrix all at once (see *Section 2* for more).

Like the *stack registers*, also each ***GP register*** can hold any object you store therein – more than just a common real number (you will learn about these other objects in *Section 2*). These *registers* are beneficial e.g. for storing intermediate results for repeated use.


GP registers
Local registers
R.98 = R210
R.97 = R209
…
…
R.02 = R114
R.01 = R113
R.00 = R112
Program steps
0000
0001
0002
…
…
9998
9999
User flags
.15 = 127
…
…
.00 = 112
K = 111
J = 110
I = CPXRES
L = LEAD0
D = SPCRES
C = CARRY
B = OVERFL
A = ALLENG
T = TRACE
Z = 102
Y = 101
X = POLAR
99
…
…
02
01
00
K = R111
J = R110
I = R109
L = R108
D = R107
C = R106
B = R105
A = R104
T = R103
Z = R102
Y = R101
X = R100
see
previous
page
R99
R98
R97
…
…
R02
R01
R00
Global registers
DRAFT

Example (with *startup default* settings):

Solve

$$\sqrt{3+\left(\frac{1.09}{1.78}\right)^2} \times \frac{ln\left[3+\left(\frac{1.09}{1.78}\right)^2\right]}{4\,cos\left[3+\left(\frac{1.09}{1.78}\right)^2\right]}$$

Solution:

First calculate the repeating term $3+\left(\frac{1.09}{1.78}\right)^2$ and store it:

1.09 [ENTER↑] **1.78** [/]	6.123 595 505 617 978×10⁻¹
[x^2] **3** [+] [STO] [K]	3.374 984 219 164 247

Then solve the entire expression, e.g. like this:

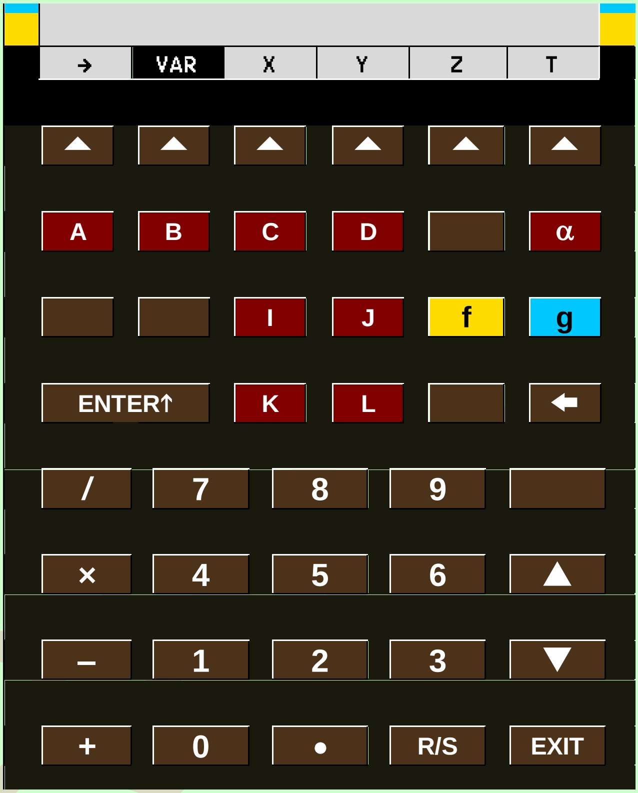
[√x]	solves the 1st factor of the expression,
[RCL] [K] [ln]	solves the numerator,
[×]	2.234 647 088 154 349
[RCL] [K] [TRI] cos	solves the 2nd part of the denominator,
[/]	2.238 529 534 683 649
4 [/]	5.596 323 836 709 123×10⁻¹

That's it – solving this expression has become really easy this way.

Variables are named storage locations. As well as each *register*, also each variable can hold any type of data (see *Section 2*).

During input processing in memory addressing, e.g. while entering parameters for storing, recalling, swapping, copying, clearing, or comparing, you will not need all the labels presented on the keyboard. Just 29 labels plus the *prefixes* will do instead. The calculator mode supporting exactly these 29 exclusively is called *temporary alpha mode* (*TAM*). As shown in examples on the next pages, it may be automatically set in memory addressing.

Entering *TAM*, the operational keyboard is temporarily reassigned as pictured overleaf.



This kind of picture is called a ***virtual keyboard*** since it may deviate from the physical keyboard of your *WP 43S*. In such a picture, **dark red** background is used to highlight changed key functionalities. White print denotes *primary* functions also on *virtual keyboards*, such as the left key in row two directly accessing (*stack*) *register* **A** in *TAM*.[45]

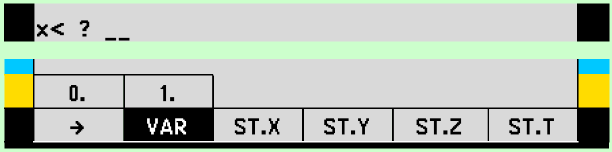
Also all other lettered *registers* can be called directly – the *stack registers* **X**, **Y**, **Z**, and **T** via unshifted *softkeys*. And accessing numbered *registers* stays as easy as can be. ➡ is for indirect addressing (see p. 60), . for local memory addresses here (see p. 233).

Variables already defined at execution time will show up in the *submenu* VAR in alphabetical order – so you can select the variable of your choice by pressing the respective *softkey*. You can also access them via α (or create new variables this way – see pp. 60f for how to do this).

Note that you will not need f or g except for *softkeys*. These may be context sensitive in *TAM*. If a comparison (e.g. x< ?) is called,[46] for instance, the f-shifted row will look like this:

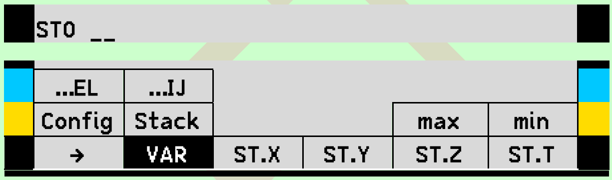
[45] What is printed white on your physical *WP 43S*, on the other hand, is called a *default primary* function.

[46] Comparisons are most useful in programming – see *Section 3*.



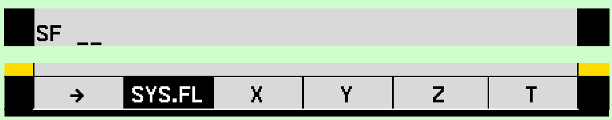
This allows for directly comparing ***x*** with the numbers 0 or 1 (see p. 60).

If STO or RCL is called, on the other hand, the shifted rows will look like this instead:



This allows for storing and recalling all your specific settings easily via STO **Config** and RCL **Config**, respectively (see p. 80). STO **Stack** stores the entire *stack* in a block of 4 or 8 *registers* (depending on *stack* size set), RCL **Stack** recalls it. And **max** (or **min**) lets you work with the maximum (or minimum) of ***x*** and the contents of the source automatically (see the *IOI*). You may press ▲ as shortcut for **f** max and ▼ for **f** min here. **...EL** and **...IJ** may be helpful when dealing with matrices (see *Section 2*).

For commands operating on *flags*, **SYS.FL** grants access to the *system flags* provided:



For all other operations asking for one trailing parameter, the *menu* will stay with a single row of *softkeys* as pictured on p. 57.[47]

---

[47] For commands operating on labels, **PROG** will be displayed instead of **VAR**, granting access to the global labels specified (see *Section 3*, *Labels*).

TAM will be terminated as soon as sufficient characters are entered for the respective operation. You may delete pending parameter input keystroke by keystroke using ⬅ and correct it if necessary – or just abort the pending command by EXIT; this will leave *TAM* immediately, returning to the mode set and the *menu* displayed before, if applicable.

If you just want to look up the current contents of a storage location without disturbing the *stack*, use VIEW.

Example:

DISP FIX 5

VIEW K returns

```
K = 3.374 98
                0.000 00
                0.000 00
                0.559 63
```

… as expected from previous example.

Note the view into *register* **K** is displayed adjusted to the left immediately below the *status bar*.

For inspecting a row of various *registers*, take RBR instead; press STATUS (or FLAGS STATUS) for checking the status of all *flags* (RBR and STATUS are explained in *Section 5* from p. 261 on).

☞ You are granted unlimited access to all the global *registers* and *user flags* allocated; there are no safety constraints like '*memory protection*' on your *WP 43S*. You are the sole and undisputed master of its memory. Thus, it is also your responsibility to take care of it – keep suitable records to avoid inadvertently overwriting or deleting your precious data.[48]

☞ You will not get 10 000 program steps and 212 *registers* and 128 *user flags* all together at the same time – see the *ReM*, *App. B,* for the reasons and for resource management.

---

48 In *Section 3*, you will learn about a method preventing your programmed routines from interfering with data of other programs.

Addressing Tables

Parameterized Comparisons:

1 User input	TEST x< ?, x≤ ?, x= ?, x≈ ?, x≠ ?, x≥ ?, x> ?				
Echo	**OP _ ?** (with *TAM* set), e.g. x<_?				
2 User input	0. or 1.	*Stack* or *lettered register* (i.e. ST.Y - ST.T, A - D, L, I - K) *or variable defined* [49]	*Register number* (range as specified on p. 63)	→ opens indirect addressing	α [50] turns on *alpha input mode* (see pp. 193ff) for a (new) *variable name*
Echo	**OP *n* ?** e.g. x=0.?	**OP? *x*** e.g. x≥ ? ST.Y	**OP? *nn*** e.g. x≠ ? r23	**OP? → _**	**OP? '_**
3 User input	Compares ***x*** with the number **0**.	Compares ***x*** with the content of *stack register* **Y**.	Compares ***x*** with the content of **R23**.	See overleaf and p. 64 for more about indirect addressing.	*Variable name* (see overleaf for more)
Echo					**OP? '*xx*'** e.g. x> ? 'ST1'

Compares ***x*** with the content of the variable called '**ST1**'. Press g TEST x> ? α S T f 1 ENTER↑ for this.

---

49 It is recommended calling variables being already defined via VAR instead of keying them in using α.

50 Note you can skip pressing f here (cf. the *virtual keyboard* above).

Register operations (requiring just a register or variable trailing):

1 User input	RCL, STO, VIEW, x≷, y≷, z≷, t≷, DEC, DSE, DSL, DSZ, INC, ISE, ISG, ISZ, etc.			
Echo	**OP _** (with *TAM* set), e.g. `RCL _` [51]			
2 User input	*Stack* or *lettered register* (i.e. ST.X - K) or *variable defined* [49]	*Register number* (range as specified on p. 63)	→ opens indirect addressing where applicable (see p. 64 and the *IOI*)	α [50] turns on *alpha input mode* (see pp. 193ff) for a (new) *variable name*
Echo	**OP *x*** e.g. `DEC K`	**OP *nn*** e.g. `VIEW 10`	**OP → _**	**OP '_**
3 User input		*Register number* (range as specified on p. 63)	*Stack or lettered register or variable defined* [49]	*Variable name* (up to 7 characters incl. one letter at least)[52]
Echo		**OP → *nn*** e.g. `STO →45`	**OP → *x*** e.g. `x≷ →L`	**OP '*xx*'** e.g. `INC 'Zähler1'`

Decrement ***k***.

Stores ***x*** in the location where ***r45*** is pointing to (see p. 64).

Swaps ***x*** and the content of the register where ***l*** is pointing to.

Increments the variable called **Zähler1**.

---

[51] For RCL and STO only, any of the keys +, −, ×, /, ▲, or ▼ may precede step 2 here. Entering such a key twice will cancel it (e.g. RCL / / equals RCL). See the chapter after next chapter for more about this *store and recall arithmetic*.

Such operators are not allowed in RCL `Config` (calling RCLCFG), RCL `Stack` (calling RCLS), RCLEL, RCLIJ, and the corresponding store operations, however.

[52] This name must be unique. If a variable with this name is not defined at execution time yet, it will be created automatically, containing zero initially.

Clearing an individual *register* or variable is most easily done by storing zero in it. Deleting a variable from memory is demonstrated on pp. 290f.

Other operations requiring one trailing parameter:

<table>
<tr><td>1 User input</td><td colspan="3">ALL, FIX, SCI, ENG, ASR, CB, FB, SB, BestF, CF, FF, SF, DSTACK, ERR, LocR, PAUSE, RM, SIM_EQ, TDISP, TONE, WSIZE, →INT, bit and flag tests, etc. (see the IOI for a complete list)</td></tr>
<tr><td>Echo</td><td colspan="3">OP _ (with TAM set),
e.g. FIX _</td></tr>
<tr><td>2 User input</td><td>Lettered flag
(i.e. X - Z, T, A - D, L, I - K)
or system flag or variable defined,[53]
if applicable</td><td>Flag number or number of bit(s) or decimals (see p. 63 for valid ranges) or any other number applicable</td><td>→
opens indirect addressing where applicable (see p. 64 and the IOI)</td></tr>
<tr><td>Echo</td><td>OP x
e.g. SF J</td><td>OP nn
e.g. SCI 12</td><td>OP → _</td></tr>
<tr><td></td><td>Sets flag 110.</td><td></td><td></td></tr>
<tr><td>3 User input</td><td></td><td>Register number
(range as specified on p. 63)</td><td>Stack or lettered register or variable defined [49]</td></tr>
<tr><td>Echo</td><td></td><td>OP → nn
e.g.
DSTACK →12</td><td>OP → x
e.g.
FIX →A</td></tr>
<tr><td></td><td></td><td>Shows as many stack levels as specified in R12 (see p. 64).</td><td>Sets fix point format with # of decimals stored in A.</td></tr>
</table>

[53] Where applicable, it is recommended calling *system flags* via SYS.FL or their short-cuts, and variables already defined via VAR, instead of keying them in using α.

	Valid number range[54]		
Registers	0 … 99	for direct addressing of global numbered *registers*	
	.0 … **.**98	for direct addressing of local *registers*	upper limits depend on current allocation
	0 … 210	for indirect addressing (≤111 without local *registers*)	upper limits depend on current allocation
Flags	0 … 99	for direct addressing of global numbered *user flags*	
	.0 … **.**15	for direct addressing of local *user flags* if allocated	
	0 … 127	for indirect addressing (≤111 without local *user flags*)	
Decimals	0 … 15	(entering any digit except 0 or 1 will terminate waiting for a further digit and close input)	
Integer bases	2 … 16	(entering any digit except 0 or 1 will terminate waiting for a further digit and close input)	
Bit numbers	1 … 64		
Word size	1 … 64 *bits*		

Please see the *ReM* for all other parameters and their valid ranges, as well as for a list of all *system flags*.

[54] Specifying low numbers (and numeric addresses), you may key in e.g. [5] [ENTER↑] instead of [0][5].

Remember some *registers* and *user flags* may also be addressed by single letters. Variables and *system flags* are generally called by their names.

Indirect Addressing – Working with Pointers

Parameters for many functions can be specified using *indirect addressing*. I.e. rather than entering the parameter itself as part of the instruction, you may supply the *register* or variable pointing to the actual parameter.

Example:

Assume *x* = **12.3**, ***j*** = **45.67**, and ***r12*** = **8.9**. Then…

STO J

RCL → J will return 8.9 since (at the time this command is executed) **J** is containing **12.3** and thus is pointing to **R12**. And now…

FLAGS SF → X will set *flag* 8, while...

DISP FIX → X will display 8.900 000 00 showing 8 decimals.

Since the content of the *register* specified is used as a pointer to the *register* wherefrom we want to read (or whereto we want to write), this method is called indirect addressing. Each and every *register* of your *WP 43S* can be used for indirect addressing.[55] And each and every register can be accessed this way (also the *stack*). Indirect addressing is most beneficial in programs when the parameter for a function is calculated (see *Section 3*, also for examples).

Store and Recall Arithmetic

As mentioned in footnote 51 on p. 61, arithmetic (and two conditional picks, i.e. *max* or *min*) can be performed upon the contents of *registers* or variables by pressing STO or RCL followed by the respective operator key (+, −, ×, /, ▲, or ▼) trailed in turn by the address or name of the storage space.

[55] Several vintage calculators, on the opposite, featured just a single dedicated *register* for indirect addressing if at all. See the *HP-34C* or *HP-15C*, for instance.

Example for store arithmetic:

123.4

STO − K closes input and subtracts **123.4** from k. The difference is stored in **K**. The *stack* and **L** remain unchanged here.

The same result could be achieved by the keystroke sequence

123.4

RCL K

x⇄y

−

STO K

RCL L but that is far clumsier (replacing one step by five) and would cost one *stack register* in addition.

The **general rule for store arithmetic** reads:

$$\text{new content of the } \textit{register} \text{ or variable specified} = \text{old content of this } \textit{register} \text{ or variable} \begin{Bmatrix} + \\ - \\ \times \\ / \\ max \\ min \end{Bmatrix} x$$

Example (from the *HP-67 OHPG*):

During harvest, farmer Flem Snopes trucks tomatoes to the cannery for three days. On Monday and Tuesday he hauls loads of 25 *tons*, 27 *tons*, 19 *tons*, and 23 *tons*, for which the cannery pays him $55 per *ton*. On Wednesday the price rises to $57.50 per *ton*, and Snopes ships loads of 26 *tons* and 28 *tons*. If the cannery deducts 2% of the price on Monday and Tuesday because of blight on the tomatoes, and 3% of the price on Wednesday, what is Snopes' total net income?

Solution:

DISP FIX 2

25 ENTER↑ 27 +		Total of Monday's & Tuesday's
19 + 23 +	94.00	tonnage
55 ×	5 170.00	Gross amount for these days
STO J	5 170.00	Take **J** for accounting

2 FIN %	103.40	Deduction for these days
STO − J	103.40	Subtracted from the total in **J**
26 ENTER↑ 28 +	54.00	Wednesday's tonnage
57.5 ×	3 105.00	Gross amount for Wednesday
STO + J	3 105.00	Added to the total in **J**
3 %	93.15	Deduction for Wednesday
STO − J	103.40	Subtracted from the total in **J**
RCL J	8078.45	Snopes' total net income from his tomatoes

Example for recall arithmetic:

78.91

RCL / 2 3 closes numeric input and divides **78.91** by ***r23***. This operation is performed in **X** alone. **L** is loaded with **78.91**. The rest of the *stack* and **R23** stay unchanged.

Alternatively, the same result could be achieved by the sequence

78.91

RCL 2 3

/ but that would replace one step by two and also cost one additional *stack register*. And **L** would differ here.

G**eneral rule for recall arithmetic**:

$$\text{new } \boldsymbol{x} \quad = \quad \text{old } \boldsymbol{x} \begin{Bmatrix} + \\ - \\ \times \\ / \\ max \\ min \end{Bmatrix}$$ content of the *register* or variable specified

Stack-wise, both store and recall arithmetic work like *monadic* functions. Note these functions may operate on each and every *register* or variable provided, also on the *stack* and even on **L**. Indirect addressing may be used as well. See pp. 218ff for more examples and advantages and the *IOI* for further details.

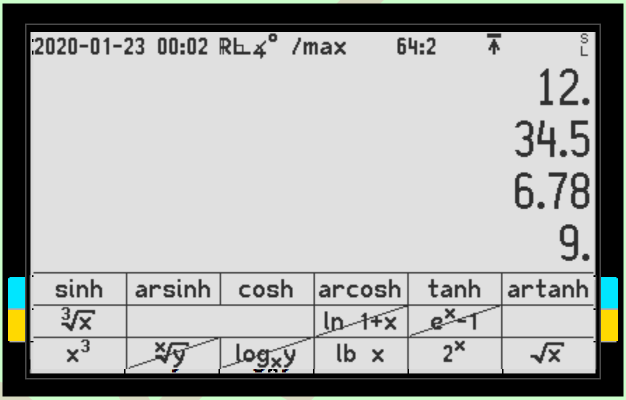
Although these techniques have been more important in times when program memory was very limited, they may be still beneficial today.

SECTION 2: DEALING WITH VARIOUS OBJECTS AND DATA TYPES

Some Display Basics

The screen is your window to your *WP 43S* – there you see what is going on and what the current results are. Going top down, you find ...

- the *status bar*,
- space for up to four rows of *standard numeric output* (and more – see points 1 to 4 below), and
- the *menu section* displaying up to three rows of *softkeys* (cf. pp. 27f).



The numeric rows deserve some additional explanations first – the *status bar* will be covered further below:

1. The **left** side of the **top (T) numeric row** is also used for output of VIEW (cf. p. 59) and SHOW (see the *IOI*) and for echoing command input until completed, i.e. until all the required command parameters are entered and the command can be executed. *Prefixes* (like f and g) will be displayed (using f and g) until they are resolved (if, however, you pressed f or g erroneously, recovery is as easy as f f = g g = NOP). And you may edit any pending operation character by character using ← or cancel it by EXIT (cf. p. 59).

2. The **left** side of the **Z numeric row** is used for displaying any error message or the output of a binary test, if applicable. Then, pending command input will stay in the top numeric row.

3. The **left** side of the **Y numeric row** is used for displaying additional (temporary) information heading ***y***, if applicable.

4. The **left** side of the **bottom (X) numeric row** is used for...

   a. echoing numeric or alphanumeric input (see pp. 25 and 193ff). Note it can take up to 42 digits, a sign, and a radix mark in *startup default* numeric format or some 40 alphanumeric characters. You may edit pending input character by character using ⬅. Numeric input will be checked and interpreted as soon as it is completed and closed, according to the calculator settings at closure time.

   b. showing additional (temporary) information heading ***x***, if applicable.

In *run mode*, any information exceeding the plain contents of the *stack registers* **X**, **Y**, and **Z** is ***temporary information***.[56] It will vanish with the next keystroke you enter: pressing ⬅ or EXIT will just clear messages, returning (for DSTACK > 2) to the pure display of ***x***, ***y***, and ***z*** – any other key will be executed in addition, if applicable.

Supported Data Types

You learned how your *WP 43S* calculates with *real numbers* in *Sect. 1*. It can do more for you: it can deal with *integers*, *fractions*, and *complex numbers* as well as *angles*, *times*, and *dates* in various formats.[57]

But how shall your *WP 43S* learn about the particular meaning of your input? Some **examples** will explain (showing **X** in *startup default* format):

---

[56] If you choose less than three *stack registers* to be displayed (see DSTACK), *temporary information* will nevertheless show up at the positions mentioned above, whenever applicable. And operations resulting in multiple output rows will display their entire output independent of the DSTACK setting always.

[57] Furthermore, it can also deal with *real* and *complex vectors* and *matrices* as well as with *alphanumeric character strings*, – these *data types* are covered comprehensively in dedicated chapters further below in this section.

All *data types* provided are listed in the *ReM*.

Input	Display	Meaning
12345.678901 EXIT	12 345.678 901	*Real numbers*, see pp. 80ff
12 E **345** ENTER↑	$12.\times10^{345}$	
123.45678901 d.ms	123°45’67.89”	Sexagesimal *angle*; see pp. 125ff also for other angular formats
1234567890 ENTER↑	1 234 567 890	*Integers* of various bases, see pp. 135ff
1234567890 # H	$12\ 34\ 56\ 78\ 90_{16}$	
10100110111 # 2	$101\ 0011\ 0111_{2}$	
901.23.4567 ENTER↑	901 $^{23}/_{4}$ 567 >	*Fraction*, see pp. 151ff
12.3 CC **−4.56** EXIT	12.3−i×4.56	*Complex numbers* in rectangular or polar notation; mantissa plus exponent format is settable as well; see pp. 154ff
	12.3∡−4.56°	
1.2345678901 h.ms	1:23:45.678 901	Sexagesimal *time*, see pp. 189f
1.0203045 .d	0001-02-03	*Date*, see pp. 191f

Some of these inputs may be interpreted and displayed differently depending on particular mode settings. *Startup default* displays are printed in light blue, further widespread formats in grey fields overleaf.

	DECIM. set	DECIM. clear
GAP **4**	1 2345.6789 01	1 2345,6789 01
GAP **3**	12 345.678 901	12 345,678 901
GAP **2** GAP **1** GAP **0**	12345.678901	12345,678901
MULT× set	$12.\times10^{345}$	$12,\times10^{345}$
clear	$12.\cdot10^{345}$	$12,\cdot10^{345}$
	123°45’67.89”	123°45’67,89”
MULT×, ¬ CPXj	12.3-i×4.56	12,3-i×4,56
¬MULT×, ¬CPXj	12.3-i·4.56	12,3-i·4,56
MULT×, CPXj	12.3-j×4.56	12,3-j×4,56
¬ MULT×, CPXj	12.3-j·4.56	12,3-j·4,56
	12.3∡-4.56°	12,3∡-4,56°
	1:23:45.678 901	1:23:45,678 901
	1:23:45.678 901 a.m.	1:23:45,678 901 a.m.

Y.MD	D.MY	M.DY
0001-02-03	01.02.0304	01/02/0304

Obviously, your *WP 43S* allows for interpreting and displaying your input very flexibly. And it allows you immediately recognizing the various *data types* and format settings looking at the screen.

Now, how can you use and combine data of various types in calculations? The matrix below lists in its 1st column ten *data types* your

WP 43S supports; and it shows what will happen when you combine various objects: an object of the *DT* as indicated in one of the lean columns at right (***y***) plus or minus an object of the *DT* in column 1 (***x***) will return an object of the *DT* at the intersection (thus, wherever a *DT* number is printed at the intersection, the corresponding combination is legal for addition or subtraction).

x	*DT* and meaning	***y*** **1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	ℤ *Long integer*	**1**	2	3	4	5	6	7	-	-	1
2	ℝ *Real number*	2	**2**	3	4	5	6	7	-	-	2
3	ℂ *Complex number*	3	3	**3**	-	-	-	7	-	-	3
4	*Angle* (in various formats)[58]	4	4	-	**4**	-	-	7	-	-	4
5	*Time* interval (in H.MS)	5	5	-	-	**5**	-	7	-	-	-
6	*Date* (in various formats)	6	6	-	-	-	**1**[59]	7	-	-	-
7	*Alpha string*[60]	-	-	-	-	-	-	**7**	-	-	-
8	*Real* matrix or vector	-	-	-	-	-	-	7	**8**	9	-
9	*Complex* matrix or vector	-	-	-	-	-	-	7	9	**9**	-
10	*Short integer*	1	2	3	4	-	-	7	-	-	**10**[61]

Example:

A *complex number* (*DT* 3) plus or minus a *real number* (*DT* 2) will result in a *complex number*.

---

[58] Angular output is tagged according to the current *angular display mode* chosen.

[59] A *date* minus a *date* returns an integer number of days (there are no other arithmetic operations on two *dates*). And a *date* plus a *real number* takes the integer part of that number and adds the respective number of days to said *date*.

[60] In additive operations on *alpha strings*, such a string must be present in **Y** at the beginning. Adding corresponds to appending ***x*** (converted to a string according to the display format set at execution time, if applicable) to string ***y***. Adding a matrix appends its abbreviation (e.g. `[3×4 ℂ matrix]`, see the chapters about vectors and matrices below). Subtractions from strings are not allowed.

[61] If *short integers* of different bases are combined by an arithmetic operation, output will be a *short integer* of the base given in **Y**.

The following matrix shows the resulting *data types* of <u>products</u> and <u>ratios</u> in the same way (note that *dates* and *alpha strings* cannot be multiplied or divided):

	An object ***y*** of *DT* …							
	1	**2**	**3**	**4**	**5**	**8**	**9**	**10**
… **times** an object ***x*** of the *DT* below returns a product of the *DT* printed at the intersection.								
1 ℤ *Long integer*	**1**							
2 ℝ *Real number*	2	**2**						
3 ℂ *Complex number*	3	3	**3**					
4 *Angle*	4	4	-	**-**				
5 *Time* interval	5	5	-	-	**-**			
8 *Real* matrix or vector	8	8	9	-	-	**8**		
9 *Complex* matrix or vector	9	9	9	-	-	9	**9**	
10 *Short integer*	1	2	3	4	5	8	9	**10**[61]
… **divided by** an object ***x*** of the *DT* below returns a ratio of the *DT* printed at the intersection.								
1 ℤ *Long integer* [62]	**1/2**	2	3	4	5	8	9	10
2 ℝ *Real number*	2	**2**	3	4	5	8	9	2
3 ℂ *Complex number*	3	3	**3**	-	-	9	9	3
4 *Angle*	-	-	-	**2**	-	-	-	-
5 *Time* interval	-	-	-	-	**2**	-	-	-
8 *Real* matrix [63]	8	8	9	-	-	**8**	9	8
9 *Complex* matrix [63]	9	9	9	-	-	9	**9**	9
10 *Short integer*	1	2	3	4	5	8	9	**10**[61]

[62] For example, 15 / 3 returns 5 while 14 / 5 returns 2.8.

[63] The matrix x must be invertible. Dividing by x is equivalent to multiplying times x^{-1}. (see the chapter *Vectors and Matrices: Calculating* below).

This is for powers:

	Any number $y > 0$ of *DT* …		
… **raised to a power** of $x > 0$ of the *DT* below returns a result of the *DT* printed at the intersection.	**1**	**2**	**10**
1 ℤ *Long integer*	**1**	2	10
2 ℝ *Real number*	1 / 2 [64]	**2**	10 / 2
10 *Short integer*	1	2	**10** [61]
… **raised to a power** of $x < 0$ … returns ...			
1 ℤ *Long integer* and **10** *short integer*		2	
2 ℝ *Real number*		2	

		Any number $y < 0$ of *DT* …		
… **raised to a power** of $x > 0$ of … returns …		**1**	**2**	**10**
1 ℤ *Long integer*		**1**	2	10
2 ℝ *Real number*	FP(x) = 0	1	**2**	10
	else	3	3	3
10 *Short integer*		1	2	**10** [61]
… **raised to a power** of $x < 0$ … returns ...				
1 ℤ *Long integer* and **10** *short integer*			2	
2 ℝ *Real number*	FP(x) = 0		2	
	else		3	

Any numbers of *data type* 1, 2, or 10 raised to *complex* powers will return *complex numbers*, as well as any *complex numbers* raised to arbitrary powers. – Other powers – involving *data types* 4, 5, 6, 8, or 9 – are not supported.

---

[64] For example, 16^(1/4) = 2 but 16^(1/3) = 2.591 8… and 16^(1/2) = 4. Compare the matrix for divisions.

Furthermore, this is for integer divisions and remainders:

… **IDIVR-divided by** an object x of the *data type* below returns an integer ratio in $\mathbf{X}$ and a remainder in $\mathbf{Y}$ of the *data types* printed at the intersection.	An object y of *data type* … **1**	**2**	**10**
1 ℤ *Long integer*	**1; 1**	1; 2	1; 10
2 ℝ *Real number*	1; 2	**1; 2**	1; 2
10 *Short integer*	1; 1	1; 2	**10; 10**[61]

Additionally, explicit type conversions are available where necessary:

An object x of *data type* … **1**	**2**	**4** *angle*	**5** *time*	**6** *date*	**10**	… will be converted in an object x of the *data type* below by the command printed at the intersection.
-	IP	-	-	-	-	**1** ℤ *Long integer*
→REAL (press .d)						**2** ℝ *Real number*
...			-	-	-	**4** *Angle*
→INT (press #)		-	-	-	→INT	**10** *Short integer* (optionally of another base)

Recognizing Calculator Settings and Status

As seen above, radix marks and gap settings are recognized in the numeric display immediately; so are date and time display modes (Y.MD / D.MY / M.DY and CLK24 / CLK12) in the time string top left within the *status bar*. Also *program-entry mode* (*PEM*) is easily recognized (see pp. 202ff).

Further modes and system states as well as many settings for specific *data types* are indicated in the *status bar*: The following specific characters may appear trailing the date and time string there, listed from left to right in various groups – indicators shown in *startup default* are printed in a light blue field again: [65]

Indicator	Set by	Deleted by	Explanation, remarks
ℂ	CPXRES	¬ CPXRES	With CPXRES set, *complex* results of *real number* calculations are allowed, like $\sqrt{-1}$. Else a domain error would be thrown in such a case (see the *ReM*, *App. C*).
ℝ	¬ CPXRES	CPXRES	
∟	¬ POLAR	POLAR	Rectangular or polar notation chosen for displaying *complex numbers*.
⊙	POLAR	¬ POLAR	
∡°	DEG	setting any other *ADM*	Current *angular display mode* (*ADM*) setting: decimal *degrees*, *grades* or *gon*, *radians*, *multiples of π*, and sexagesimal *degrees*.
∡g	GRAD		
∡r	RAD		
∡π	MULπ		
∡"	d.ms		
/max or /2345	DENANY	Can only be modified by DENMAX	Fraction display settings. The current value of the maximum displayable denominator is shown behind the fraction bar (*startup default* and absolute maximum is 9999, displayed as /max). With DENANY clear, DENFIX toggles a specific character trailing DENMAX in the *status bar*.
/2345f	DENFIX & ¬ DENANY	¬ DENFIX, DENANY	
/2345× or /2345·	¬ DENFIX & ¬ DENANY	DENFIX, DENANY	

[65] The symbol ¬ means "not", i.e. the trailing *system flag* cleared, while "&" denotes a logical "and" and a comma a logical "or" in this table.

Indicator	Set by	Deleted by	Explanation, remarks
64:1	1COMPL	setting any other *integer sign mode* (*ISM*)	Settings for *short integers*. First two digits tell the *word* size, the character after the colon the *ISM*. *Startup default* is 64 *bits* (the maximum) and 2's complement. CARRY and OVERFLOW may trail the *ISM* but are only lit if set.
64:2	2COMPL		
64:u	UNSIGN		
64:s	SIGNMT		
A	α, ALPHA; ▲ if α is set	pressing EXIT in *AIM* unless in a *menu*, ¬ALPHA	*Alpha input mode* (*AIM*) is set. Upper (A) or lower (α) case letters can be entered now.
α	▼ if A is set		
	program waiting for user input	program running	Will also be lit if a program is stopped by EXIT or R/S – then will be cleared by next keystroke.
	see remarks	*WP 43S* idling	**Flashes** while a program is running; **steady** while a function is executing.
	top of program memory	else	Program pointer at step 0000.
	timer running in background	idle timer	See the TIMER (or stopwatch) application on pp. 263f.
	serial I/O in progress	idle communication line	See *Serial Input and Output of Data and Programs* on pp. 233f.
	data are being sent to printer		
U	USER	USER	Toggles *user mode* (see pp. 292f).
	low battery	battery voltage > 2.5 V	A low battery will reduce processor speed automatically. Your *WP 43S* will shut off when voltage drops < 2.0 V.

The *startup default configuration* is indicated in a *status bar* like this:

2017-05-08 23:49 RL∡° /max 64:2 ↟

On the other hand, choosing 12h time format (or M.DY), setting CPXRES, FRACT, DENFIX and a four-digit DENMAX, selecting unsigned *short integers*, setting CARRY and OVERFLOW, having a program waiting for input with *AIM* set, timer and printer running in background, *user mode* set, and a low battery would be reflected in the following *status bar*:

05/08/17 11:49pm CL∡ʳ /3546f 64:u A

Note also ⌛ and ⇕ might show up at right end of the *status bar*.

Getting Special Information: RBR, STATUS, VERS, etc.

Some commands and tools use the display in a special way. These operations are listed below:

1. The *Matrix Editor* is described comprehensively on pp. 163ff.
2. RBR allows for browsing the contents of all *registers* currently allocated (see pp. 261ff).
3. STATUS (or FLAGS STATUS) returns free space available, memory currently used, *user* and *system flags* set (see pp. 263f).
4. TIMER calls the timer or stopwatch application (see pp. 264ff).
5. FBR browses all the characters defined in the fonts provided.

Further commands throw *temporary information* as defined on p. 68:

1. ERR and MSG display the corresponding error message. See the *IOI* and *App. C* of the *ReM* for more.
2. r, VIEW, x̂, and ŷ return results headed by text.
3. Commands returning two or three values at once (like →P, R←, DATE→, DECOMP, x̄, s, L.R., SUM, M.DIM?, RCLIJ, Σ+ and Σ−) tag their output (see e.g. pp. 20 and 109f).

4. VERS generates a string showing version and build of the firmware running on your *WP 43S* (WHO works in a similar way):

```
WP 43S v0.1 b0123  by Pauli, Walter & Martin
```

A few far-reaching commands (like CLALL, CLPALL, or RESET) ask you for confirmation before executing. Answer either Yes by pressing 3 (or ENTER↑ or XEQ) or No by pressing 7 (or EXIT or ←); any other input will be ignored. Note that such an action explicitly confirmed cannot be undone by ↶.

Localising Numeric Output

You can summon display preferences for *reals*, *times*, and *dates* all at once according to your region's customs and practices using dedicated commands (all contained in DISP). In the table starting overleaf, …

- **radix mark** denotes the decimal separator;
- **GAP** states the digit group interval – after ***n*** digits a narrow blank is displayed (cf. examples on p. 70); this follows *ISO 80000-1*.[66]
- **JG** states the year the *Gregorian calendar* was introduced in the particular region, typically replacing the *Julian calendar* (or national calendars in East Asia);[67]
- background colors are chosen as on pp. 69f.

Most people using radix commas employ multiplication dots while those using radix points need a cross for multiplication to avoid misunder-

[66] As far as we know, the *WP 43S* is the first pocket calculator displaying numbers the way internationally agreed on. Previous calculators featuring limited displays had to use e.g. points or commas as crutches since they could not display narrow blanks .

[67] Officially, the *Gregorian calendar* became effective at 1582-10-15 in the catholic world. Many states and territories switched later for various reasons (check the dates in *Wikipedia*). You can enter the date applicable at your location using J/G (see the *IOI* for this command). Note there are still other calendars widespread, e.g. in the Muslim world. See also the chapter *Dates* below.

standings. This latter convention causes further ambiguities in vector multiplication (see pp. 174ff).

Com-mand	GAP	Radix mark[68]	Time	Date[69]	JG	Remarks
SETCHN	**4** [70]	point	24h	Y.MD	1949	
SETEUR	**3**	comma	24h	D.MY	1582	Also applies to South America (and – with other JGs – to Indonesia, South Africa, the area of the former Soviet Union, and Vietnam).
SETIND	**3** [71]	point	24h	D.MY	1752	Also applies to India, Pakistan, Nepal, Bhutan, Myanmar, Bangladesh, and Sri Lanka.
SETJPN	**3**	point	24h	Y.MD	1873	
SETUK	**3**	point	12h	D.MY	1752	Also applies to Australia and New Zealand.[72]
SETUSA	**3**	point	12h	M.DY	1752	

---

[68] See *https://en.wikipedia.org/wiki/Decimal_separator* for a world map of radix mark use. Looks like an even score in this matter. Thus, the international standard *ISO 80000-1* allows either a decimal point or a comma as radix mark and requires a narrow blank as unambiguous separator of digit groups (it explicitly states that points or commas shall not be used as group separators to avoid ambiguity).

[69] See *https://en.wikipedia.org/wiki/Date_format_by_country* also for a world map of date formats used. The international standard *ISO 8601* states Y.MD for dates and 24h for times. This combination is common in East Asia (see SETCHN and SETJPN).

[70] Chinese counting and traditional mathematics work with powers of 10 000 while (originally Indian, then Persian, then) European counting and mathematics work with powers of 1000. Thus, Chinese count using intervals 一 (= 1), 十 (= 10), 百 (= 100), 千 (= 1000), 万 (= 10 000), 十万 (= 10 × 10 000), 百万 (= 100 × 10 000), 千万 (= 1000 × 10 000), 亿 (= 10^8), 十亿 (= 10^9), 百亿 (= 10^{10}), 千亿 (= 10^{11}), etc. The command GAP **4** takes care of this notation while GAP **3** formats the European way.

[71] Proper South Asian (a.k.a. Indian) formatting would require separators every two digits for numbers over thousand. Think of *lakh* = 10^5 and *crore* = 10^7. Actually, an amount of 50 cr. Rupees ($= 5 \times 10^8$) reads 50,00,00,000 Rs. in Indian newspapers.

[72] 24h is taking over in the UK, so SETIND will work there then as well.

Note that the following settings and formats can be stored collectively at one location: entire decimal display format (see next chapter), *angular display mode*, *date* and *time* display settings, parameters of integer and fraction display modes, curve fit model chosen, rounding mode, and the status of all *system flags*. STOCFG stores this *configuration* in the *register* or variable you specify.[73] RCLCFG recalls such information and will set (or reset) your *WP 43S* accordingly.

Real Numbers: Changing the Display Format

As mentioned in *Section 1*, the numbers you calculate with (decimal numbers or measured values) are *reals* frequently. Any number you enter containing one [.] and/or an [E] is interpreted by your *WP 43S* as a *real number* unless there is additional information given (cf. pp. 68f). The majority of functions provided by your *WP 43S* operate on *reals*.

As soon as input of a *real number* is closed, its mantissa will be displayed right adjusted as far as possible (cf. p. 25). *Startup default* format (ALL **0**) shows all digits of the number if less than 16 are needed to do so. Your *WP 43S* will automatically turn to mantissa plus exponent format (cf. pp. 25f) if more than 15 digits are needed.[74]

There are two flavors of the latter format: SCI and ENG. SCI is called scientific notation. ENG looks almost like SCI but the exponent will always be a multiple of three, corresponding to the *SI* unit prefixes – thus it is called the ENGineer's notation (see examples below).

You can choose whether ALL shall turn either to SCI or to ENG. And you can define the switch point from ALL to SCI or ENG by specifying a positive parameter for ALL (telling up to how many decimal zeros you allow before the output shall be switched):

---

[73] Actually, it stores even more – see *Section 6.*

[74] No matter what display format or notation you select, these rounding options affect the display only. Your *WP 43S* continues using its full precision (typically 34 digits for *real numbers*) internally always; this can be displayed by SHOW until next keystroke.

Example (beginning with *startup default* settings):

Input:	Display:
–700	-700_
1/x	-1.428 571 428 571 429×10^{-3}
DISP ALL 3	-0.001 428 571 428 571
10 /	-1.428 571 428 571 429×10^{-4}
FLAGS SF SYS.FL ALLENG	-142.857 142 857 142 9×10^{-6}
▼ ALL 4	-0.000 142 857 142 857
10 /	-14.285 714 285 714 29×10^{-6}
CF SYS.FL ALLENG	-1.428 571 428 571 429×10^{-5}

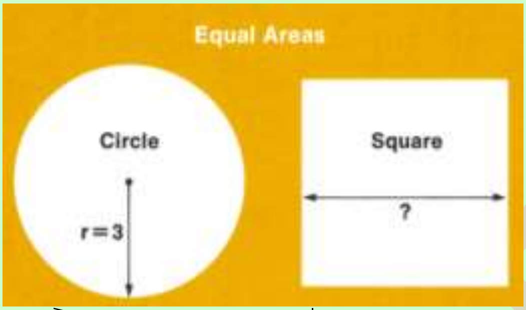
There is one more format provided: FIX. With FIX, the radix mark is set at a fixed position on the screen and stays there (a.k.a. fixed point notation); it floats in the other formats – see the examples below.[75]

You can specify the number of decimals you want to see with SCI, FIX, or ENG (note the parameter of FIX and SCI specifies the number of decimals to be shown while the parameter of ENG specifies the total number of digits displayed within the mantissa minus one):

Format / Input	*Startup default* format (ALL **00**, SCIOVR)	FIX **5**	SCI **5**
107.12345678 ENTER↑	107.123 456 78	107.123 46	1.071 23×10^{2}
1/x **2** ×	1.867 004 725 311 852×10^{-2}	0.018 67	1.867 00×10^{-2}

See more examples of displays varying according to popular choices for GAP, decimal radix mark, and multiplication symbol (cf. the examples shown on p. 70):

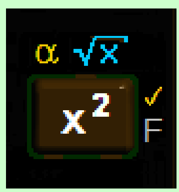
[75] Deviating from previous calculators, output of ×10^{0} is suppressed on your *WP 43S*.



Input \ Format	FIX **3**	ENG **6**
89012345678.9 +/- ENTER↑	-89 012 345 678.900	$-89.012\ 35{\times}10^9$
	-89 012 345 678,900	$-89{,}012\ 35{\cdot}10^9$
	-890 1234 5678.900	$-89.0123\ 5{\times}10^9$

Nearly all functions for real number display format control are found in DISP: FIX, SCI, ENG, ALL, GAP, rounding, and more. Please see the *ReM*.

Real Numbers: Squares and Cubes and their Roots



You find x^2 and $\sqrt{x}$ on the keyboard of your *WP 43S*, while x^3 and $\sqrt[3]{x}$ are in EXP (cf. p. 27). The following **example** using these four functions contains some of the most popular problems of antique mathematics:

What size square has the same area as a circle whose radius is 3 arbitrary units? And what size cube has the same volume as a sphere whose radius is 3 again? And what can we tell about their surface areas?

Solutions:

The area of a circle is $A_C = \pi r^2$. The area of a square is $A_{sq} = a^2$. The volume of a sphere is $V_S = \frac{4}{3}\pi r^3$, while its surface is $A_S = 4\pi r^2$. And the volume of a cube is $V_{cu} = a^3$, while its surface is $A_{cu} = 6a^2$.

Thus,

DISP FIX 3

3 x^2 π ×	returns	28.274	for the area of the circle. Then
$\sqrt{x}$	returns	5.317	for the edge length of the square.

Furthermore,

3 EXP x^3 π ×			
4 × 3 /	returns	113.097,	the volume of the sphere. Then
$\sqrt[3]{x}$	returns	4.836	for the edge length of the cube with same volume. Thus,
x^2 6 ×	returns	140.320	for the surface of the cube.

Finally,

3 x^2 π × 4 ×	returns	113.097	for the surface of the sphere.

Actually, there was no necessity calculating this last surface here – why?

Here a little winter sports problem of our time:

Example:

Chuck Carver swings down a ski run with moderate 30 km/h. The curvature of his skis allows for turns with 12 m radius. He claims carving this way without any sliding on an almost flat part of the run. If true then how many g he had to withstand there? Can we believe his story?

Solution:

The centrifugal force is $F_c = r\omega^2 m = 2\pi \frac{v^2}{r} m$, thus the corresponding acceleration is $a_c = 2\pi \, {v^2}/{r}$. In consequence, the total acceleration acting along *Chuck's* body axis is $a_T = \sqrt{g^2 + a_c^2}$. Measured in multiples of g, this means ${}^{a_T}/_{g} = \sqrt{1 + \left({}^{a_c}/_{g}\right)^2}$.

DISP FIX 0 1		
30 E 3 ENTER↑	30 000.0	
3600 /	8.3	Chuck's speed in $^{\mathrm{m}}/_{\mathrm{s}}$
x^2	69.4	
12 / 2 × π ×	36.4	
CONST $g_\oplus$ /	3.7	
x^2 1 + $\sqrt{x}$	3.8	meaning 3.8 g.

Even if this might be possible to stand shortly for a young sportsman like *Chuck*, the snow under him can hardly bear the corresponding forces – it will break so *Chuck* will inevitably slide in a greater radius leading to less acceleration.

Another problem, found in a calculator manual of 1976:

Example:

Finding himself floating dangerously close to the jagged peaks of the Canadian Rockies, intrepid balloonist *Chauncy Donn* frantically cranks open the helium valve on his spherical balloon. Gas from the helium tank increases the balloon's radius from 7.5 *meters* to 8.25 *meters*.[76] *Donn* clears the mountain tops safely. How much did the volume of the balloon increase?

Solution:

Since the volume of a sphere is $V = \frac{4}{3}\pi r^3$, the difference of two such volumes is $\Delta V = \frac{4}{3}\pi(r_2^3 - r_1^3)$. One decimal shall do.

[76] In the *HP-21 Owner's Handbook*, the balloonist *Ike Daedalus* had to increase the radius from 25 to 27 *feet* in 1975. Sometimes some progress was observable.

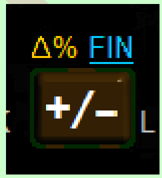
8.25 EXP x³ — 561.5
7.5 x³ − — 139.6
π × — 438.7
4 × 3 / returns 584.9 m^3 for the volume increase.

Real Numbers: Percent Change

Δ% calculates the percentage of change from ***y*** to ***x***.

Example (continued from above):

This is a volume increase of how many percent?



Solution:

7.5 x³ — 421.9
8.25 x³ — 561.5
Δ% returns 33.1 % increase.

Another example:

How about designing an almost optimum bicycle gearing for hilly areas? Feel free to choose sprockets and gear clusters to your liking.

Solution:

As long as drag may be neglected, an optimum gearing will show equal velocity ratios between subsequent gears (or uniform increase of distances per crank revolution). There are several ways you can reach this, depending on the number of sprockets chosen at front and rear.

One inexpensive way is taking three front sprockets of 48, 36, and 24 teeth and getting a standard seven-gear cluster featuring 13, 15, 17, 20, 23, 26, and 30 teeth at the rear. This will result in the following distances travelled per crank revolution (***d/r$_c$*** in *meter*) for a 26" MTB:

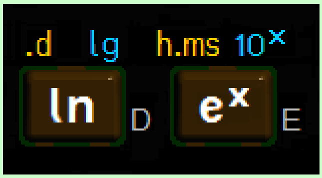
Gear	1	2	3	4	5	6	7	8	9	10	11	12
Front	24				36			48				
Rear	30	26	23	20	26	23	20	23	20	17	15	13

d / r_c	1.66	1.92	2.17	2.49	2.87	3.25	3.73	4.33	4.98	5.86	6.64	7.66
Δ%	-	15.7	13.0	15.3	15.3	13.2	14.8	16.1	15.0	17.7	13.3	15.4

Assuming you pedal with 60 rpm constantly, such a bicycle will cover velocities between 6 and 28 km/h (or up to 37 km/h for 80 rpm). Using also some statistical functions provided on your *WP 43S* (i.e. Σ+, x̄, and s explained on pp. 99ff), you will determine a mean speed increase per gear step of (15.0 ± 1,4)%, being quite uniform and convenient for town and country.[77] Feel free to try other configurations.

Real Numbers: Logarithms and Powers (a.k.a. Antilogs)

Your *WP 43S* features two logarithmic functions on its keyboard and two more in EXP (cf. p. 27):



ln calculates the *natural logarithm* of ***x***, i.e. the logarithm of ***x*** to the base **e** (being Euler's constant, see CONST). Thus, ln inverts eˣ.

lg returns the (*common*) *decadic logarithm*, i.e. the logarithm of ***x*** to the base 10. lg inverts 10ˣ.[78]

lb x calculates the *binary logarithm*, i.e. the logarithm of ***x*** to the base 2. lb x inverts 2ˣ.

[77] Note that you will get just 12 out of 3×7 theoretically possible gears this way. This is due to gear overlaps; and you will want to avoid extreme chain skew for sake of chain life. On the other hand, if you plan for a recumbent bike, the latter restriction might not apply anymore. Then you may think about a combination of three sprockets with a seven-gear cluster leading to 17 different, usable gears following the so-called "half-step-and-granny" scheme; speed increase in half-step range will be 9% per gear step; this gearing will cover velocities from 5 to over 40 km/h.

For detailed specifications as well as pictures, graphics, diagrams, tables, and further information about gearing bicycles yourself, please order "*Die Fahrradschaltung*" (144 pages written in German) written by the same author – just contact me.

[78] You may be used to a calculator label LOG for the decadic logarithm; though this is a mathematically ambiguous notation or worse, so we avoided it (cf. *ISO 80000*, 2-12.5 and 2-12.6).

LOG$_x$y is the most general of these four functions: it returns the logarithm of y to the base x. LOG$_x$y can be used to invert y^x.

The operating manual of the world's very first electronic pocket calculator featuring transcendental functions, the *HP-35* (cf. p. 53), presented just a single example concerning this then new class of pocket-able functions:

Example:

Suppose you wish to use an ordinary barometer as an altimeter. After measuring the sea level pressure (30 inches of mercury) you climb until the barometer indicates 9.4 inches of mercury. How high are you? Although the exact relationship of pressure and altitude is a function of many factors, **a reasonable approximation** is given by:[79]

$$\frac{altitude}{[feet]} = 25\,000 \times ln\left(\frac{30}{pressure/[inches\ of\ Hg]}\right)$$

Solution:

DISP FIX 0 0 should suffice here.

30 ENTER↑ 9.4 /

ln

25 E 3 × returns 29 012. .

[We suspect that you may be on *Mt. Everest* (29 028 *feet*).]

Note the concise and factual style of this text. The *HP-35* was a calculator made by engineers for engineers, and the manual was alike. – This example was reprinted in the *HP-45 OH*. Thereafter, it underwent slight modifications:

Example (from the *HP-21 OH*):

Having lost most of his equipment in a blinding snowstorm, ace explorer *Buford Eugobanks* is using an ordinary barometer as an altimeter. After measuring the sea level pressure (30 *inches of mercury*) he climbs until

[79] Emphases in these quoted examples were added by me.

the barometer indicates 9.4 *inches of mercury*. Although the exact relationship of pressure and altitude is a function of many factors, *Eugobanks* knows that **an approximation** is given by the formula …

This problem remained in subsequent calculator manuals though the explorers changed for unknown reason. A picture of the scenery was added in 1976, and not every snowstorm was worth mentioning anymore. Then, however, a switch of units reached the *Himalayas* – and also the weather and the methods changed:

Example (in *Solving Problems with Your Hewlett-Packard Calculator* of 1978):

With most of his equipment lost in an avalanche, mountaineer *Wallace Quagmire* must use an ordinary barometer as an altimeter. Knowing the pressure at sea level is 760 *mm of mercury*, *Quagmire* continues his ascent until the barometer indicates 238 *mm of mercury*. Although the exact relationship of pressure and altitude is a function of many factors, *Quagmire* knows that **an approximation** is given by the formula:

$$\frac{altitude}{[\mathrm{m}]} = 7\,620 \times ln\left(\frac{760}{pressure/[mm\ of\ Hg]}\right)$$

Where is *Wallace Quagmire*?

Solution:

760 [ENTER↑] **238** [/]

[ln] **7620** [×] returns `8 847.`

Quagmire appears to be near the summit of *Mt. Everest* (8 848 m).

And it seems neither he nor his barometer returned from this expedition since this example did neither show up in the *HP-41C OHPG* nor later anymore. Perhaps there was something wrong with the recalibration of his instrument? [80]

By the way, the altitude approximation formula for standard *SI* units reads:

$$\frac{altitude}{[\mathrm{m}]} = 7\,620 \times ln\left(\frac{1\,013}{pressure/[\mathrm{mbar}]}\right) = 7\,620 \times ln\left(\frac{101\,300}{pressure/[\mathrm{Pa}]}\right)$$

Beyond the barometric scale, there are more logarithmic scales used in science and engineering, e.g.

- in astronomy for assessing the brightness of stars or
- in chemistry for the power of acids (pH); most popular may be
- the *decibel* (dB) in acoustics and electronics (see <u>U→</u>, pp. 276f) and
- the so-called *upwardly unlimited Richter scale* for magnitudes of earthquakes.[81]

Example:

One of the strongest earthquakes observed recently was the one causing the devastating tsunami in the Indian Ocean (near Indonesia) in December 2004. It had a magnitude of 9.1. Another one near Japan in March 2011 – with a magnitude of 9.0 – led to another tsunami and the

---

[80] Maybe <u>this</u> is the reason why the last three countries on this planet do not switch to *SI* – do they fear the recalibrations inevitably necessary for their measuring equipment?

[81] This name is still popular in the news although not quite true anymore. The actual moment magnitude scale for earthquakes differs but is logarithmic as well.

'*Fukushima nuclear accident*'. Compare with the '*great San Francisco earthquake*' of 1906 with a magnitude of 7.9.

Solution:

The formula for comparing the energies released in two different earthquakes (with their magnitudes known) reads

$$\frac{E_2}{E_1} = 10^{1.5(M_2 - M_1)}$$

Again, no decimals are needed here – we can continue with the display settings as they are:

9.1 ENTER↑ **7.9** −
1.5 × 10^x returns **63.** and
9 ENTER↑ **7.9** −
1.5 × 10^x returns **45.** .

So the energy released in said Japanese earthquake in 2011 was 45 times greater than the so-called '*great San Francisco earthquake*'. And said earthquake in the Indian Ocean was even 63 times more intense.

Taking into account that published magnitudes of earthquakes never show more than one decimal, we did not lose anything real setting the *WP 43S* to FIX **0** here.

Even small numeric differences will gain significance when raised to powers. Human brains are not well equipped for such operations, so we recommend taking good care in such cases.

Example:

What difference in magnitude will cause double destruction?

Solution:

Rewriting the formula above results in $\Delta M = \frac{2}{3} lg\left(\frac{E_2}{E_1}\right)$. Thus, for double destruction we need a magnitude difference of

DISP FIX 0 1
2 lg **2** × **3** / equalling **0.2** only.

But there are also friendlier applications of logarithms:

Example:

How many bits are required if the unsigned integer 3.7×10^9 shall be the maximum to be handled by a microprocessor?

Solution:

3.7 E **9** lb x returns 31.8 , so 32 *bits* will suffice.

If we had a tri-state logic, however,

RCL L **3** $\log_x y$ returned 20.1 , so 21 cells would suffice.

Providing y^x, your *WP 43S* also allows for raising any positive *real number* to an arbitrary *real* power, as well as any negative *real number* to an arbitrary integer power, all returning *real* results. Compare e.g. the *Mach number* formula on p. 45.

In combination with 1/x, y^x also provides a simple way to extract roots:

Example (with *startup default* settings):

What is the fifth root of 17 ?

Solution:

This is equivalent to $17^{1/5}$, so **17** ENTER↑ **5** 1/x y^x will do.
This solution path may be faster accessed and executed than the alternative **17** ENTER↑ **5** g EXP $\sqrt[x]{y}$. Both keystroke sequences, however, will return 1.762 340 347 832 317.

Let's return to *Fukushima* for a final and (alas!) more down-to-earth application of powers and logs:

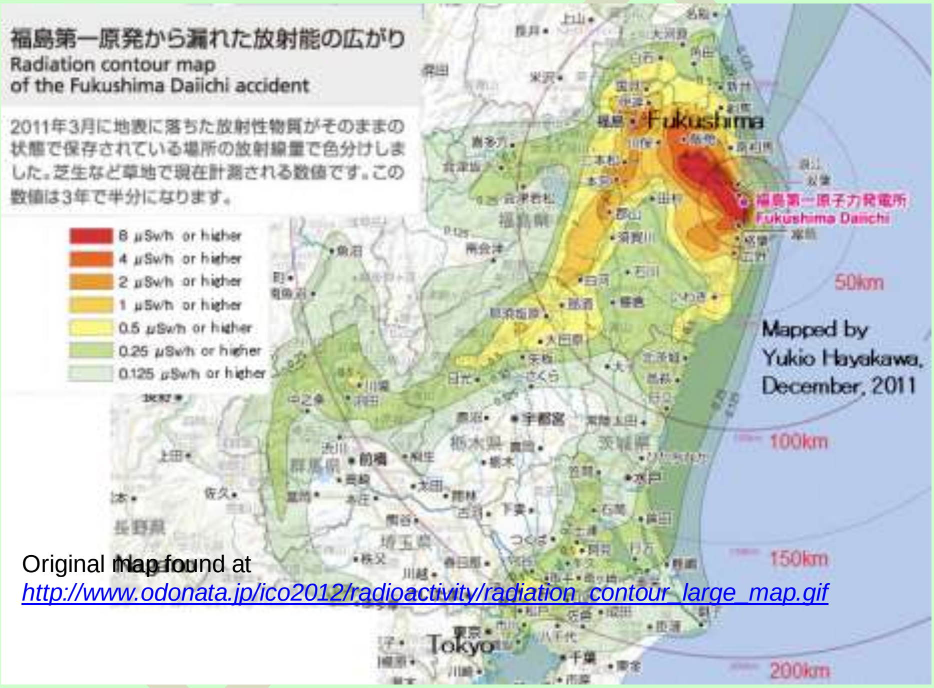
Example:

Locations in a distance of 30 km to the nuclear plant being devastated by the tsunami in March 2011 showed radioactivity in the soil of some 1...3 $\mathrm{MBq/m^2}$ corresponding to an annual radiation dose of 4 mSv in 2013 (see the map). Assume this was mainly caused by ^{137}Cs then; this radioactive caesium isotope has a half-life of 30.2 years.[82]

To the best of our knowledge today, an unborn child must not receive a dose of more than 1 mSv before birth. So when will it be reasonably safe to let the evacuated inhabitants of the villages in that area return to their homes finally?

Solution:

Assuming there will be no further nuclear accident there, the isotopes set



Original map found at
http://www.odonata.jp/ico2012/radioactivity/radiation_contour_large_map.gif

free will stubbornly decay following the inevitable laws of physics. Having had a radioactivity $\boldsymbol{a_0}$ at time zero, the activity $\boldsymbol{a}$ at an arbitrary later time $\boldsymbol{t}$ will be

$$a = a_0 \times 2^{-\left(t/T_{1/2}\right)} . \text{ Hence, } \quad t = T_{1/2} \times lb\left(\frac{a_0}{a}\right).$$

[82] With a probability of 94%, ^{137}Cs decays emitting an electron with a kinetic energy of up to 512 keV plus a γ-ray of 662 keV. These facts are just for your information – they do not affect the calculation here.

1 mSv in nine months corresponds to an annual dose of $^4/_3$ mSv. Well, the 2013 annual dose of 4 divided by $^4/_3$ equals 3, and

DISP FIX 0 0
3 EXP lb ×
30.2 × returns 48. years.

So you can recommend reproductive people shall rather not live in that area earlier than 2061. Senior inhabitants may return far sooner.[83]

Quite similar considerations apply to nuclear waste of power plants – at the bottom line, there are many tons of radioactive material produced decaying with half-lifes exceeding thousand years; and this means you

---

[83] Note that different limits are considered 'reasonably safe' for the public by different national authorities. By nature, all such limits are arbitrary to some extent since we talk about probabilities here, and there are no step functions in probability but smooth transitions (cf. the chapter after next chapter). Furthermore, a large fraction of world-wide knowledge about damage caused by radiation in human bodies in the long range is still based on extrapolation of experiences collected since 1945 following two large-scale events in Japan (and 67 more near Bikini until 1958). Another experiment well known was started in the USSR in 1986 – Belarus and the Ukraine have to bear the consequences until today. Mankind knows of the physics of radioactivity for some 120 years only so far, that's short!

Note there are further risks linked to agriculture in the area around Fukushima – they are beyond the scope of this simple sample calculation though.

Also note this example covers a worst case scenario. Actually, radioactivity is washed to deeper layers of soil with time, reducing the activity seen at the surface. And there are mitigation efforts in the area (many km^2): at some places the contamination was washed off houses and trees, and the top layers of contaminated soil were removed, storing them in big black plastic bags 'elsewhere'; 2.3 million m^3 of soil are deposited there already, 12 million more are expected by the authorities – an area of 1.6 km^2 is provided for 'interim storage'. Cost of disposal is going to be 1.9×10^{12} ¥ (estimated by the administration in 2019). Furthermore, 1.1 million m^3 of contaminated water are stored separately – no idea yet where that shall go (see *Frankfurter Allgemeine Zeitung of 2019-03-10*). I guess where it will go, though…

Success of all these mitigation efforts may reduce the waiting time calculated above; failure will not extend it at least. Today is too early for a definitive assessment – we still know too little about long term effects. None of these efforts, however, can ever reduce the given natural half-lifes of the radioactive isotopes set free and spread in this nuclear accident.

have to 'put them away' safely for really long times – a task kept under wraps for decades but not solved by waiting so far.[84]

The formula above is a nice example of a mathematically simple law of physics linking science and society quite closely.

Real Numbers: Hyperbolic Functions

Hyperbolic functions tell us something about free hanging ropes, cables, chains, and the like. Your *WP 43S* provides three hyperbolic functions and their inverses in the g-shifted row of EXP (see p. 27) and TRI:

sinh	*Hyperbolic sine*.	arsinh	Inverse *hyperbolic sine*.
cosh	*Hyperbolic cosine.*	arcosh	Inverse *hyperbolic cosine*.
tanh	*Hyperbolic tangent*.	artanh	Inverse *hyperbolic tangent*.

We found the following in the *HP-32 OH*[85] though we modified it a bit:

---

[84] Surprise! Mankind has absolutely no experience with locking something away reliably for several thousand years (look at the pyramids of Gizeh, for instance). Note the material must also be tagged properly (KEEP OFF!) in a way staying readable and comprehensible for all that time – zero experience either.

Sad example**:** A huge concrete coffin holds almost 90 million *liters* (equivalent to 90 000 m^3) of US nuclear waste on the Marshall Islands (remember Bikini). Now sea-level rise (caused by *anthropogenic global warming*) is eating away at the dome, and the USA is not interested in helping the tiny Pacific Ocean republic to do anything about it (see the *Los Angeles Times of 2019-11-10*).

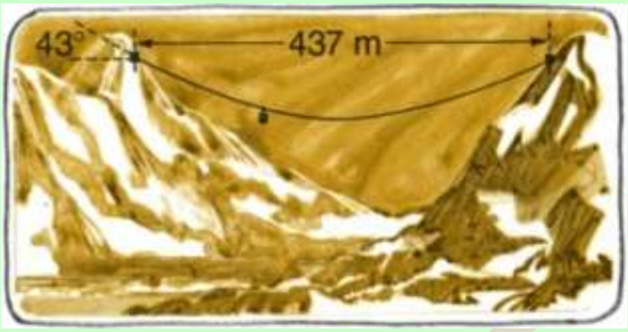
Sometimes you might meet people talking about 'transmuting' that entire long-living radioactive waste by converting it to isotopes with significantly shorter half-lifes by some nuclear reactions (never met anybody being more specific in this matter so far). If that would be physically possible for all that material, however, the energy needed for that transmutation process would easily outweigh the energy 'produced' by nuclear power plants before. As a matter of fact, the companies who made profits with those power plants for decades are very reluctant in definitely solving the waste problem they created so far.

As far as mankind knows today, prospective fusion plants will not produce any long-lived isotopes in operation. Wait and watch.

[85] This was *HP*'s first pocket calculator featuring hyperbolic functions. It was launched in 1978. Note that the *SR50* of *Texas Instruments* (*HP's* arch rival in those years of the so-called 'calculator wars') provided hyperbolic functions four years earlier already.

Example:

In *Upper Lagunia*, a tram[86] carries tourists between two peaks in the *Baruvian Alps* that are the same height and 437 *meters* apart. How long does it take the tram to travel from one peak to the other if it moves along its cable at 135 *meters per minute*? Before the tram latches onto the cable, the angle from the horizontal to the cable at its point of attachment is found to be 43°.



Solution:

The travel time is given by the formula

$$t = \frac{d}{v} \times \frac{\tan\alpha}{arsinh(\tan\alpha)}.$$

Let's set

DISP FIX 2 since we do not need more decimals displayed.

Then

43 TRI tan

ENTER↑ duplicates this intermediate result on *stack* for numerator and denominator.

arsinh /

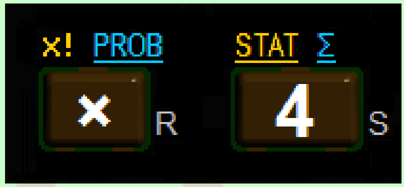
437 × 489.30 m is the length of the cable.

135 / 3.62, i.e. a bit more than 3 ½ *minutes.*

[86] Translator's note: British readers might frown here at least.

Real Numbers: Probabilities – Factorials, Combinations, Permutations, and Distributions

Besides the keyboard commands **Δ%** and **x!**, you find a lot of probability and statistical operations in your *WP 43S*, going far beyond the *Gaussian distribution*. It contains all the preprogrammed functions implemented in *WP 34S* and more – presumably the maximum set available in a pocket calculator world-wide. These operations are stored in the adjacent *menus* PROB and STAT.



PROB includes also the functions for *combinations* and *permutations*.

Example (from the *HP-32 OH*):

Willie's Widget Works wants to take photographs of its product line for advertising. How many different ways can the photographer arrange their eight widget models?

Solution:

The total number of possible arrangements possible is given by the *factorial* $8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 8!$

8 **x!** returns 40 320 for this number.

Example (continued):

The photographer looks through his view-finder (in 1978) and decides that he can show only five widgets if his camera is to capture the intricate details of the widgets … How many different sets of five widgets can he select from the eight?

Solution:

The number of sets equals the number of possible *combinations* (i.e. the number of possible different sets of y different objects taken in quantities of x objects at a time; no object appears more than once in a set, and different orders of the same x objects are not counted separately here):

8 **ENTER↑** **5**

PROB C_{yx} returns 56 for the number of sets.

Example (continued):

Again, there are different arrangements feasible. How many pictures of different widget arrangements are possible within these limits?

Solution:

The number of possible arrangements is 5! according to the statement above. Thus,

5 x! returns 120 for that number. And

× returns 6 720 for the number of significantly different pictures.

This is the number of possible *permutations* of 5 items out of 8 (i.e. the number of possible different arrangements of y different objects taken in quantities of x objects at a time; no object appears more than once in an arrangement, and different orders of the same x objects are counted separately here).[87] It can be obtained in one step by keying in

8 ENTER↑ **5** P_{yx} returning 6 720 .

Furthermore, PROB contains ten continuous and five discrete *distributions* for calculating probabilities, confidence intervals, etc.[88] These functions share a few features:

[87] These challenging tasks changed the photographer significantly within one year.

[88] In a nutshell, discrete statistical distributions deal with "events" governed by a known mathematical model. Such statistical events may be persons entering a store, radioactive nuclei decaying, faulty parts appearing, etc. The *PMF* then tells the probability to observe a certain number of such events, e.g. 7. And the *CDF* gives the probability to observe up to 7 such events, but not more.

For doing statistics with continuous statistical variables – e.g. the heights of three-year-old toddlers – similar rules apply: Assume we know the applicable mathematical model; then the respective *CDF* gives the probability for their heights being less than an arbitrary limit, for example less than 1 m. And the corresponding *PDF* tells how these heights are distributed in a sample of let's say 1000 kids of this age.

BEWARE: This is a very rudimentary sketch of this topic only – turn to a good textbook to learn dealing with statistics properly.

Translator's note for German readers: *PMF* und *PDF* entsprechen der *Wahrscheinlichkeitsdichte*, *CDF* der *Verteilungsfunktion* bzw. *Wahrscheinlichkeitsverteilung*.

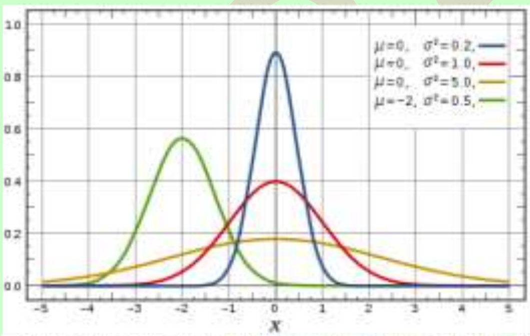
- Discrete distributions (like *Poisson*, *binomial*, *negative binomial*, *geometric*, and *hypergeometric*) are confined to integers. Whenever your *WP 43S* sums up a *probability mass function* (*PMF*) $p(n)$ to get a *cumulated distribution function* (*CDF*) $P(m)$, it starts at $n = 0$. Thus,

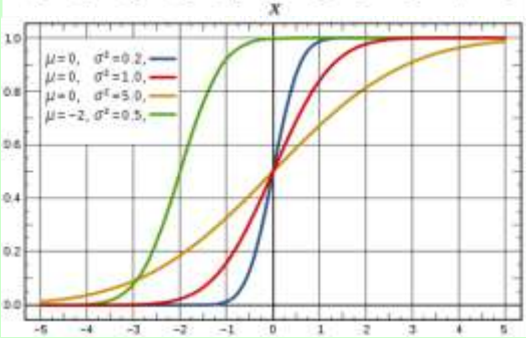
$$P(m) = \sum_{n=0}^{m} p(n)$$

- Continuous distributions (like *Cauchy, exponential, logistic, log-normal,* two kinds of *normal*, *Fisher's F, Student's t, Weibull*, and *chi-square*) operate on *reals*. Whenever your *WP 43S* integrates a function, it starts at left end of the integration interval. Thus, integrating a continuous *probability density function* (*PDF*) $f(x)$ to get a *CDF* works as

$$P(x) = \int_{-\infty}^{x} f(\xi) d\xi$$

- Many frequently used continuous *PDF*s look more or less like the ones plotted in the upper diagram overleaf. The lower diagram shows their corresponding *CDF*s, using the same scale and colors.





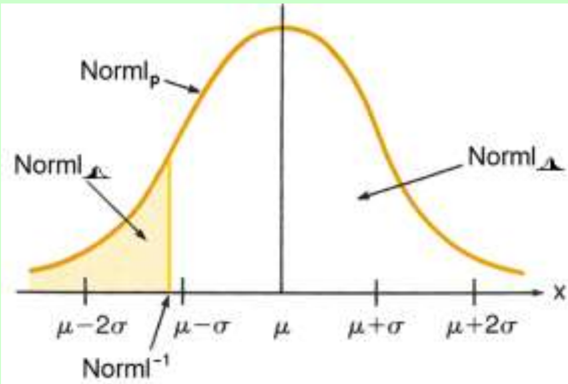
Typically, any *CDF* starts at 0 with a slope of almost zero, becomes steeper then, and runs out at 1 with its slope returning to zero. This holds even if the respective *PDF* does not look as nicely symmetric as the sample *normal distributions* plotted here.

Thus, obviously you will get the most precise results for the *CDF* on its left side using P. On its right side, however, where P slowly approaches 1, the *error probability* $Q = 1 - P$ will be more precise. Thus, also the right sided Q

is computed in your *WP 43S* for each distribution, independently of P. Definitions are:

- for discrete distributions: $$Q(m) = \sum_{n=m}^{\infty} p(n)$$
- for continuous distributions: $$Q(x) = \int_x^{\infty} f(\xi) d\xi$$

- With an arbitrary *CDF, e.g.* NORML◣ (returning P), you will find the name NORML◢ used for the function returning Q, NORML^{-1} for the inverse of the *CDF* (the so-called *quantile function*), and NORML$_P$ for its *PDF* on your *WP 43S*. This naming scheme applies also to the ***binom**ial*, ***Cauch**y* (a.k.a. *Lorentz* or *Breit-Wigner*), ***expon**ential*, *Fisher's* ***F***, ***geom**etric*, ***hyper**geometric*, ***log-norm**al*, ***logis**tic*, ***nega**tive* ***bin**omial*, ***Poiss**on*, *Student's* ***t***, and ***Weibu**ll* distributions. The *Chi-square* distribution is denoted differently following mathematical tradition. See <u>PROB</u> on p. 111 or the *ReM*.



Find application examples of distributions in the next two chapters.

Real Numbers: From Probability to Statistics – Accumulating Data, Calculating Means, Standard Deviations, and Confidence Limits; Curve Fitting, Forecasting, and Checking Dices

There is also a wealth of commands for sample and population statistics in <u>STAT</u>, applicable in one or two dimensions. After clearing the summation *registers* by **CLΣ** initially, use **Σ+** to accumulate your experimental data (typically counted or measured values); weighted data require the weight in **Y**, pairs of data or coordinates of data points shall be entered in **X** and **Y**. [Σ–] is provided for easy data correction.

Data analysis functions are found in STAT as well: e.g. *arithmetic mean* $\bar{x}$, *sample* and *population standard deviations* s and σ, and *standard error* s_m (a.k.a. *standard deviation of the mean*).

Example:

Archibald is champion of the *Golden Bow*, his archers club. In his standard exercise, aiming at a target disk of 1.5 m diameter at a distance of 50 m, his arrows scatter symmetrically around the center of the target showing quite a small variance. Actually, *Archibald's* statistics tells his arrows have a standard deviation (*SD*) of 1 *foot* at that distance. Assume his shots are distributed normally around the center of the disk, how often must he walk further than 50*m* to collect an arrow?[89]

Solution:

[DISP] FIX [3]		
0 [STO] [I]		Archibald's mean = center of disk.
1 [U→] x: feet→m	0.305	, 1 *foot* in *meters*, Archibald's *SD*.
[STO] [J] [STO] [0] [1]		store this *SD* for later re-use.
1.5 [ENTER↑] 2 [/]	0.750	, the radius of the target disk.
[PROB] Norml: Norml⊿	0.007	, the error probability.
2 [×] [1/x]	72.102	so *Archibald* has to collect an arrow in the green only once in 72 shots on long term average.

Example (continued):

One of his buddies and competitors, *Bill*, also sends his arrows to the same target disk with his hits scattering symmetrically around the center of said disk, too. He, however, has to pick up about one out of 15 arrows in the green on average. What is his *SD* in the target plane?

[89] Many of our customers live in a country where long range weapons play a significantly greater role than in most civilized societies, hence this explanatory example. Foreigners travelling through that country, watch out! Please note that we refrained from using firearms here, though our resistance was strained almost to the limit.

Solution:

Keys	Display	Comment
15 [1/x]	0.067	, i.e. about 7% of *Bill's* arrows miss the target disk.
2 [/]	0.033	~3% misses on either side.
1 [STO] [J]		to get the *standard normal* distribution.
Norml^{-1}	-1.834	, the corresponding lower limit of this distribution.
[+/-] .75 [x≷y] [/]	0.409	m = *Bill's SD*. Just store it since we will need it again soon:
[STO] [0] [0]		
[RCL] [0] [1]		Note that the *SD* of *Archibald's* arrows is just...
[Δ%]	-25.47	% narrower than *Bill's*, but his rate of misses is more than 10 times less.

There are applications of this methodology in industry, where the scattering (a.k.a. variation, variance) of a production process is compared with its tolerance limits. Resulting from such comparisons, so-called *capability indices* are computed, directly linked to the amount of scrap to be expected in the process investigated. Please consult applicable literature and standards – look for *process capability*.

On the other hand, we may continue with our example as is, guiding you to advanced statistics:

Example (continued):

Bill quietly practiced in a *Zen* cloister during his summer vacation. Returning, he went to the *Golden Bow* immediately on next weekend and sent 50 arrows to his club's standard disk. Only two missed, with one of them scratching the very edge of the disk. Cheers! But is this just a lucky chance success (within the usual scattering of results to be expected) or probably a consequence of his extra training efforts?

Solution:

Calculate *Bill's* new *SD*:

Keys	Display	Comment
1.5 [ENTER↑] 50 [/]	0.030	
2 [/]	0.015	= 1.5% misses on either side.

Norml^{-1} -2.170 , the corresponding lower limit of the standardized normal distribution.

+/_ .75 x≷y / 0.346 m = *Bill's* new *SD*.

Now, is this *significantly* better than his previous *SD*? Statisticians have found it is better (based on a *confidence level* of 95%) if it is lower than the 95% *confidence limit* of his old *SD*. We assume his old *SD* (s_o) was computed based on 60 shots. Then the formula for the *single-sided lower 95% confidence limit* of this old *SD* reads:

$$\sigma_L = s_o \times \sqrt{\frac{59}{\left(\chi^2_{59;\,0.95}\right)^{-1}}}$$

The expression in the denominator is the *inverse chi-square* for 95% probability and 59 *degrees of freedom*. Calculate inside out as usual:

59 STO J		the degrees of freedom must be stored in **J**.
.95 χ^2: $(\chi^2)^{-1}$		calls the inverse chi-square, returning
	77.931	.
/	0.757	
√x	0.870	
RCL × 0 0	0.356	m for σ_L.

Looks like *Bill's* training made a difference!

Well … within 95% confidence. If we had required 99% confidence instead, the lower *confidence limit* had been 0.337 m (you can easily verify this now) – then *Bill's* new weekend result would have been an insufficient indicator for a *significant* improvement.[90]

STAT contains also functions for curve fitting, featuring ten different regression models (linear, exponential, logarithmic, power, root,

[90] Applying statistics may cause that you might have more doubts than without – but such is life: doubts increase with knowledge. Only very dumb people have no doubts and may easily feel great therefore.

Generally, standard *confidence limits* and *levels* (also those defined for indicating *significant differences*) may depend on the country or industry or science you are working in. Note the term *significant* is well defined in statistics – this definition may deviate from common language. Be sure to check the applicable valid standards before blindly copying the exemplary calculations demonstrated in this manual.

hyperbolic, and more – see the *ReM*), their parameters, the forecasting functions $\hat{x}$ and $\hat{y}$, and the *coefficient of correlation* r. The fit model applied will be displayed heading numeric output after any command related to fitting (i.e. after CORR, COV, L.R., s_{XY}, $\hat{x}$, and $\hat{y}$). And after L.R., even the generic formula of the regression model applied will be shown (see examples below).

The command BESTF tells your *WP 43S* to select the regression model fitting your data 'best' (i.e. resulting in the largest absolute *coefficient of correlation*, approx. 1). Then, an elevated asterisk (*) will trail the name of the fit model chosen this way automatically. Like with all other auto-functionality, you should know what you are doing here.

Example (from the *HP-27 OH*):

If *Galileo* had wished to investigate quantitatively the relationship between the time (***t***) for a falling object to hit the ground and the height (***h***) it has fallen, he might have released a rock[91] from various levels of the *Tower of Pisa* (which was leaning even then) and timed its descent by counting his pulse. The following data are measurements *Galileo* might have made:

***t* (pulses)**	2	2.5	3.5	4	4.5[92]
***h* (Pisan *feet*)**	30	50	90	130	150

Unlike *Galileo*, you are equipped with a *WP 43S*; so what can you learn from this experiment? Let's look what we may find:

DISP FIX 4

STAT

CLΣ	$\bar{x}_G$	ε	ε_p	ε_m	PLOT
Σ-	$\bar{x}_w$	s_w	σ_w	s_{mw}	
Σ+	$\bar{x}$	s	σ	s_m	SUM

CLΣ

[91] I hope not! A pebble would have done as well if not better.

[92] These raw data really do not look very plausible, and actually it is dubious whether *Galileo* made such experiments using the *Tower of Pisa* at all, but at least *HP* believed that its calculator customers would believe in that story in 1976.

30 ENTER↑ **2**

Σ+ returns

```
                                 0.355 9
Data point 001                  30.000 0
                                 2.000 0
```

Note that **Σ+** takes ***x*** and ***y***, adds them to the statistical sums, increments the count of data points, and gives you feedback (note this output contains *temporary information* as explained on p. 68). Your next input after **Σ+** will overwrite ***x*** : [93]

50 ENTER↑ **2.5** **Σ+**
90 ENTER↑ **3.5** **Σ+**
130 ENTER↑ **4** **Σ+**
150 ENTER↑ **4.5** **Σ+**

```
Data point 005                 150.000 0
                                 4.500 0
```

▼

GaussF	CauchF	ParabF	HypF	RootF	
LinF	ExpF	LogF	PowerF		BestF

BestF **0** instructs your *WP 43S* to pick the curve fit model matching these experimental data best (as explained above).

▼

	$\bar{x}_{RMS}$	x_{max}	x_{min}		OrthoF
	$\bar{x}_H$				
L.R.	r	s_{xy}	cov	$\hat{x}$	$\hat{y}$

L.R.

[93] Remember Σ+ disables *stack lift*. Though note that accumulation of 2D data will slowly overwrite the *stack*.

		4.500 0
Power*	a_1 =	1.994 0
$y = a_0 x\verb|^|a_1$	a_0 =	7.722 6

Your *WP 43S* chose power regression as the model fitting these given data best. Let's check the *correlation coefficient*:

r	returns	Power* 0.997 6

This is an almost perfect correlation. The equation expressing the experimental results best is hence $h \approx 7.72 \times t^{1.99}$ (with ***t*** measured in *pulses* and ***h*** in *Pisan feet*). *Galileo* could not know around 1600 yet, but we know today that $h = \frac{1}{2}\, g\, t^2$.

The task to determine the size of a *Pisan foot* and *Galileo's* heartbeat frequency is left for the reader.

In addition, we found the following linear regression **example** in various *HP* calculator owners' manuals of 1976 - 78. It reads typical for the thinking at that time:

Big *Lyle Hephaestus*, owner-operator of the *Hephaestus Oil Company*, wishes to know the slope and y-intercept of a least squares line for the consumption of motor fuel in the *United States* (*of America* [94]) against time since 1945 (in 1978!). He knows the data given in the table:

Motor fuel demand (millions of *barrels*)	696	994	1330	1512	1750	2162	2243	2382	2484
Year	1945	1950	1955	1960	1965	1970	1971	1972	1973

Solution:

Hephaestus [95] *could* draw a plot of motor fuel demand against time.

[94] Differentiating from *Los Estados Unidos Mexicanos*, for example.

[95] Maybe his ancestors emigrated from Greece: *Hephaistos* is the ancient Greek god of fire and forging (and maybe of underground natural resources as well?).

However, with his *WP 43S*, *Hephaestus* has only to key the data into the calculator using the **Σ+** key, then press **L.R.**. [96]

DISP **FIX** **2**

STAT **CLΣ**

696	ENTER↑	1945	Σ+	994	ENTER↑	1950	Σ+
1330	ENTER↑	1955	Σ+	1512	ENTER↑	1960	Σ+
1750	ENTER↑	1965	Σ+	2162	ENTER↑	1970	Σ+
2243	ENTER↑	1971	Σ+	2382	ENTER↑	1972	Σ+
2484	ENTER↑	1973	Σ+				

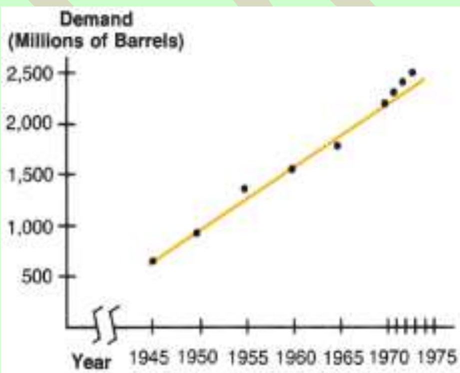
▲ **L.R.**

Linear*	$a_1 =$	61.16
$y = a_0 + a_1 x$	$a_0 =$	-118 290.63

Your *WP 43S* chose linear regression as the model fitting the given data best here. Let's check the *correlation coefficient*:

r returns Linear* 0.99

Based on this good correlation result, *Hephaestus* confirms the automatic choice and is even tempted to extrapolate the observed trend of motor fuel demand to (then) future years.



Example (continued):

If *Hephaestus* wishes to predict the demand for motor fuel for the years 1980 and 2000, he keys in the new ***x*** values and presses **ŷ**.

Similarly, to determine the year that the demand for motor fuel is expected to pass 3 500 million *barrels*, *Hephaestus* keys in **3 500** (the new value for ***y***) and presses **x̂**.

[96] The boss computes himself! And he also seems being even able to do it properly! Looks like that was a time before general managers, CEO's, and large staffs became fashionable. But see also next footnote.

1980	$\hat{y}$	returns	Linear*	2 808.63
2000	$\hat{y}$	returns	Linear*	4 031.85

These were forecasts (i.e. extrapolations based on the fit model employed) of the demands in 1980 and 2000 at that time.

3500	$\hat{x}$	returns	Linear*	1 991.30

– the demand was expected to pass 3.5 billion *barrels* in 1992. [97]

Another **example** from the *HP-27 OH*:

The *chi-square statistic* measures the goodness of fit between two sets of frequencies.[98] It's used to test whether a set of observed frequencies differs from a set of expected ones sufficiently to reject the hypothesis under which the expected frequencies were obtained.

In other words, you are testing whether discrepancies between the observed frequencies (O_i) and the expected frequencies (E_i) are significant, or whether they may reasonably be attributed to chance. The formula generally used is

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

If there is a close agreement between the observed and expected frequencies, χ^2 will be small. If the agreement is poor, χ^2 will be large.

Let's demonstrate the application of such a *chi-square statistic*[99] using the following problem, presuming *startup default* settings of your *WP 43S*:

---

[97] If he had plotted his data, *Hephaestus* should have been warned looking at his last three data points. Often, plots carry extra information which may be lost easily when dealing with numbers only. Actually, *HP's* example is not a good choice for extrapolating without plotting.

[98] 'Goodness of fit' tells us how good both sets match.

[99] Do not confuse this χ^2 defined here with the χ^2 distribution mentioned in previous chapter (and employed below very soon). They are different! Unfortunately, however, both chi-squares are called and spelled equally. It looks like the naming commission was inattentive here at the crucial time, perhaps distracted by discussing the outcome of a recent casino visit.

A suspect dice from a *Las Vegas* casino is brought to an independent testing firm to determine its bias, if any. The dice is tossed 120 times and the following results obtained:

Number	1	2	3	4	5	6
Frequency	25	17	15	23	24	16

Solution:

Expected frequency is 120/6 = 20 for each number here. For calculating χ^2, just enter:

Keystrokes	Display
DISP FIX 0 0	
25 ENTER 20 − x^2	25
17 ENTER 20 − x^2 +	34
15 ENTER 20 − x^2 +	59
23 ENTER 20 − x^2 +	68
24 ENTER 20 − x^2 +	84
16 ENTER 20 − x^2 +	100
20 /	5

Now, is this χ^2 large or small? Statisticians have found it is to be considered 'small' if χ^2 is less than the value of the inverse χ^2 *CDF* for the *degrees of freedom* (*DOF*, here $n - 1 = 5$) and the *significance level* applicable (here 5%). As seen above already, also this χ^2 function is provided in your *WP 43S*. Note that a *significance level* of 5% equals an *error probability* of 5% and a *confidence level* of 95%. Simply key in:

Keystrokes	Display	
5 STO J	5	for the *DOF*;
.95 PROB χ^2: $(\chi^2)^{-1}$	11.	

Since 5 is less than 11, χ^2 is small enough to conclude that this dice is fair (with 95% confidence).

Real Numbers: Some Industrial Problems Solved

To get an idea of further real-life opportunities covered by your *WP 43S* and of some constraints inherent to statistics, see the sample applications shown below. All of them are demonstrated employing the traditional 4-*register stack* but will work with the 8-*register stack* as well.

Application 1 (scrap rate, confidence limits):

Assume you own a little tool shop, produce axis pins in series, and want to know the quality of the parts you produce. You drew a *representative sample* of pins (all being nominally equal parts!) and precisely measured their real sizes using a proper instrument. How can you know your batch will be ok?

Example:

Ten turned pins drawn from a batch produced on a precision lathe, diameters measured: 12.356, 12.362, 12.360, 12.364, 12.340, 12.345, 12.342, 12.344, 12.355, and 12.353. From earlier large scale investigations, you know that diameters from this production process follow a *Gaussian* (or *normal*) *distribution*.

Now you should just know your objective:

- Do you want to know what pin diameters you will get in your batch? Statistics cannot tell you about <u>all</u> of them but it will tell you where to find <u>almost all</u> (e.g. 99%) of them.

Example (continued):

DISP FIX 3

STAT CLΣ

0 ENTER↑ 12.356 Σ+

```
Data point 001          0.000
                       12.356
```

Continue accumulating the remaining measured sample data:

12.362 Σ+ **12.360** Σ+ **12.364** Σ+ **12.340** Σ+
12.345 Σ+ **12.342** Σ+ **12.344** Σ+ **12.355** Σ+
12.353 Σ+

```
Data point 010          0.000
                       12.353
```

Knowing these pins are drawn from a *Gaussian* process, you get the best estimates for mean and standard deviation of your batch by pressing

$\bar{x}$

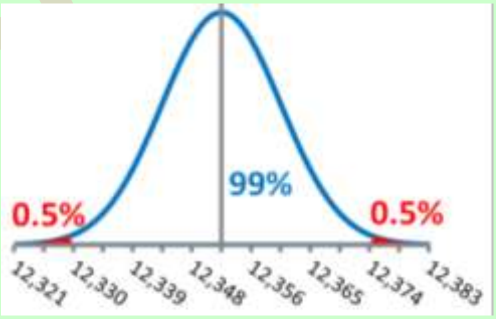
```
ȳ =                     0.000
x̄ =                    12.352
```

STO I s

```
                       12.352
sy =                    0.000
sx =                    0.009
```

STO J We stored $\bar{x}$ and s_x for the next steps already.

Now, if 99% of a batch is found inside some arbitrary symmetric limits of a *Gaussian* process then 0.5% will be out on either side since the *Gaussian distribution* is symmetric around its mean.



Thus, based on the ten pins analyzed, you may expect 0.5% of all pins with diameters less than

.005 PROB

					0.005
	NBin:	Geom:	Hyper:	Binom:	Poiss:
LgNrm:	Cauch:		Expon:	Logis:	Weibl:
Norml:	t:	C_{yx}	P_{yx}	F:	χ^2:

Norml:

Norml_p	Norml		Norml_e	Norml^{-1}

Norml^{-1}	12.330

and another 0.5% with diameters greater than

.995 Norml^{-1}	12.375

If you should observe significantly more than 0.5% of your pins beyond either limit, this indicates your process may be running out of control.

Assume the pins shall have a nominal diameter of 12.35. Then – based on this sample analysis – you can safely commit to hold a tolerance of ±0.05 (you will hardly produce any scrap as long as your process continues running the way you found it). If your customer would try, however, to force you to accept a tighter tolerance of ±0.02, you must expect some losses:

12.35 ENTER↑	12.350	
.02 −	12.330	= lower limit.
Norml	0.006	= lower scrap = 0.6%.
x≷y	12.350	
.02 +	12.370	= upper limit.
Norml	0.021	= upper scrap = 2.1%.
+	0.026	= total scrap.[100]

What will hurt you even more than these 2.6% scrap you must expect now (i.e. more than 1 out of 40 pins) will be the inevitable necessity to establish a very precise and constant sorting tool or machine to ensure only good pins will pass to your customer. Thus, stay firm (if you can afford it) and

[100] Translator's note: *Scrap* is a very interesting word to follow through various languages: Ausschuss, brak, desecho, detriti, rebut, schroot, Schrott, skrot, šrot, sucata, utskrot, брак, хлам, etc. The latter is pronounced almost like spam. Note in German and Swedish, the same word may be used for *scrap* and *committee*.

refuse that customer request to constrict your tolerance limits – it may well be you cannot afford becoming weak here.

- Are you interested in the mean pin diameter of your batch? So you know how much space you must provide to store a stack of e.g. 50 pins? Then determine the applicable mean and the size of its variation; then use them to find both upper and lower limit confining the mean with a probability of e.g. 95%.

Example (continued):

Since we have got a sample drawn out of a *Gaussian* process, the arithmetic mean is applicable, the *standard error* tells its variation, and *Student's t* is required. For the latter, we need its *degrees of freedom*. Press

STAT ▼ n	10.000	recall the number of points.
1 − STO J	9.000	store the degrees of freedom.
▲ s_m	0.003	is the standard error.

Having 95% inside means having 2.5% outside at either end (cf. previous diagram).[101] Thus, one must generally take 0.025 and 0.975 as arguments in two subsequent calculations using the *quantile function* of *t* to get both 95% limits below and above the sample result:

.025 PROB t: $t^{-1}(p)$	-2.262
×	0.006
STAT $\bar{x}$	12.352
x≷y R↓ [102]	12.352

[101] The value of 95% is called the *confidence level* of this calculation. In this example, you calculate the 95% *confidence limits* for the mean value. Instead of 95%, also 99% are frequently applied. We recommend checking the applicable valid standards before blindly copying any example calculations here. Of course, you are free to apply other confidence levels wherever they fit your needs.

Translator's note for German readers: *Confidence limit* entspricht der *Vertrauens-bereichsgrenze* und *confidence level* dem *Vertrauensniveau*.

[102] $\bar{x}$ returns $\bar{x}$ and $\bar{y}$ (as was shown above). Only $\bar{x}$ is interesting in this example, however, so pressing x≷y R↓ moves $\bar{y}$ quickly out of the way. In a program, DROPy will be a better alternative since it leaves the *stack* order as is (see. *Sect. 3*).

x≷y	0.006	
−	12.346	= lower limit.
RCL L	0.006	= last *x*.
2 × + [103]	12.358	= upper limit.

Now you know what to expect for the future average diameter of such batches. Hence a stick being

50 ×	617.900	long inside will suffice for holding 50 pins in 97.5% of all cases.

12.346 and 12.358 are the *95% confidence limits* of the mean calculated above. So here is a chance of 2.5% that the mean will be < 12.346 and an equal chance that it will be > 12.358. These chances are an inevitable consequence of the fact that you know something about a small *sample* only (drawn out of a large *population*), but want or have to tell something about said total *population*.[104] If you cannot live with these uncertainties or the widths of the confidence limits, do not blame statistics but collect more or more precise data instead.

Application 2 (quick and easy measuring system analysis):

Your colleagues in R&D have specified that particle accelerator beam pipes made of a special stainless steel shall have a magnetic susceptibility ≤ 0.01. How can you verify whether or not the susceptibility meter available in the laboratory is sufficiently precise to control the series production of those tubes?

Solution:

1. Collect 30 samples of material covering the susceptibility range you are interested in. This range could extend e.g. from 0 to about 0.015

[103] The upper *confidence limit* can be calculated this easy way since $t^{-1}(p)$ is symmetric around the mean value. Else it would have been necessary to repeat the above calculation (except the last two steps) for an input value of 0.975.

[104] Statisticians call these chances '*probabilities of a type I error*' or '*probabilities of an error of the first kind*'.

Translator's note for German readers: *Type I error* entspricht dem *Fehler 1. Art.*

here.[105] Mark each sample unambiguously (e.g. by numbering it).

2. Use the measuring instrument under investigation to measure all samples carefully under controlled conditions. Record as many decimals as possible. Write each measured value in a table next to the respective sample number.

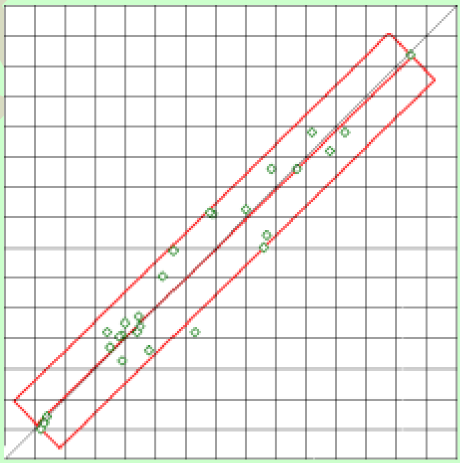
3. Measure a 2nd time under 'the same' conditions, but following another sample sequence (just shuffle the samples). Don't allow for looking at the data measured previously (hiding these data will be very helpful if you acquire the values manually)! Write each 2nd measured value behind the 1st one in the row carrying the respective sample number.

4. Get your *WP 43S*. Clear its statistical *registers* via CLR CLΣ. Then enter all 30 pairs of values using STAT Σ+ . The 1st measured value shall be y, the 2nd be x – thus, input will be

   mv1 **ENTER↑** ***mv2*** **Σ+**

   for each sample (alternatively, you can enter your statistical data into a matrix, then accumulate all points at once – see pp. 185f).

5. It is recommended to plot these 30 points. The plot shall look like an ant trail following the center line $y = x$ (see a typical scatter plot here).[106]



---

[105] Nature allows for positive susceptibilities only. Note there is no requirement to know the <u>exact</u> susceptibilities of your samples beforehand – they shall just fall in said range, cover it fairly homogenously (cf. the plot overleaf), and the samples must be resilient enough to stay constant in your measurements. No need for any investment in expensive gauges here – real life has proven scrap may well do.

[106] Even pencil plotting on quadrille paper will do. Since important here, we might implement some basic scatter plot abilities in this calculator. See the *ReM*.

Go to an expert in metrology if your diagram should deviate fundamentally from the one pictured here.

In four decades professional experience, I found such correlation diagrams being the most powerful though easy tools for assessing the quality of real-life measuring processes. Even manually drawn clean correlation diagrams will support your

6. Let your *WP 43S* fit a straight line through the points and compute

$$c_0 = \frac{T}{30 s_x s_y} \sqrt{\frac{s_x^2 + r^2 s_y^2}{1 - r^2}}$$

with ***T*** being the width of the tolerance zone you want to control. So select the orthogonal linear fit model:

▲ OrthoF	
r x² STO K	get the *coefficient of correlation* and store its square.
▼ s x²	get s_x^2.
R↓	roll it out of the way.
x² ×	get $r^2 s_y^2$.
R↑ +	return s_x^2 from the top *stack register* and calculate the numerator.
1 RCL − K	calculate the denominator.
/ √x	this is the 2nd factor now.
s × / 30 /	divide by **30** s_x s_y .
.01 ×	this returns c_0 for our exemplary ***T*** now.

If you get $c_0 \geq 1$ then this measuring device may be used for controlling series production with this tolerance zone under these conditions (i.e. it is a *capable* instrument for this control job) – else look for a more precise instrument, better measuring conditions, or a wider tolerance.

Application 3 (significant changes):

Assume you have taken a sample out of an arbitrary industrial production process at day 1. Then you have changed the process parameters, waited for stabilization, and have taken another sample of same size at day 2 (there may well have been a longer time interval between both sampling days). Being serious, you have meticulously measured and recorded a critical quantity (e.g. a characteristic dimension) for each

---

decisions far better than staring at numbers only – actually plotting is just required to verify there are no bad surprises hidden in your setup and data acquisition. With capable measuring systems, the resolution required for showing all measured points clearly separated from each other might, however, exceed the capability of a pocket calculator screen by far. The calculations will be correct nevertheless.

specimen investigated at both days. Now: do these two samples show any *significant difference*?

The following simple three-step test is well established.[107] It may easily save yourself some unwanted embarrassments in your next presentation or after your next publication:

1. Accumulate your sample data. Then let your *WP 43S* compute the means and *standard errors* for both samples, and their *normalized distance* $d = |\bar{x} - \bar{y}| \,/\, \sqrt{s_x^2 + s_y^2}$. If you are working with four *stack registers*, this calculation could look like the following:

STAT s_m	returns both standard errors in **X** and **Y**.
x^2 x≷y x^2 + $\sqrt{x}$	so this is the entire denominator.
$\bar{x}$	returns both $\bar{x}$ and $\bar{y}$.
− |x|	thus, this is the numerator
x≷y / STO D	and this is d.

2. Let your *WP 43S* calculate the critical limit t_{cr} of *Student's t* for f *degrees of freedom* and a probability of 97.5% now:

Σ n	recall the number of samples measured.
1 − STO I	calculate the *degrees of freedom* f and store them for *Student's t*.
.975 PROB t: $t^{-1}(p)$	as mentioned above, the requested *quantile function* lives in PROB. It takes the degrees of freedom stored in **I** to get t_{cr}.

If $d < t_{cr}$, the test indicates the difference between both samples is due to random deviations only. Congratulations – you have got a robust process regarding the parameters you changed!

Else continue.

3. Let your *WP 43S* compute a new critical limit t_{cs} for f and 99.5%:

.995 $t^{-1}(p)$	get t_{cs}.

If $d \geq t_{cs}$ now, then the test indicates a *significant difference*

[107] This test assumes your samples are both drawn from a *Gaussian* process which is frequently the case in real life (but shall be verified).

between both samples. Congratulations – your parameter change caused a significant effect!

Else (i.e. for $t_{cr} \leq d < t_{cs}$) you simply cannot decide seriously based on the information provided – your samples may contain too little data or your measurements were not precise enough or the process is scattering too far etc. Though do not let your audience lead you in temptation: stay silent or mumble something like "investigation in progress" at the utmost.

Application 4 (operating characteristics):

Assume you draw a sample of 20 parts out of a production batch of 100 parts and check the sample thoroughly. What is the probability *P* to find at least one random defect in such a sample if the overall probability for a defect in such a batch is 5%, 2%, or 1%?

This is a textbook example for applying the *hypergeometric distribution*. $P(\boldsymbol{n} \geq 1)$ equals $100\% - p(\boldsymbol{n} = 0)$. Thus, the solution is as simple as this:

DISP FIX 3		
100 STO I	store batch size	
20 STO K	store sample size	
0.05 STO J	store 5% overall defect probability	
0 PROB g Hyper: Hyper▲	returns	0.319
1 x≷y −	returns	0.681
0.02 STO J	store 2% overall defect probability	
0 Hyper▲	returns	0.638
1 x≷y −	returns	0.362
0.01 STO J	store 1% overall defect probability	
0 Hyper▲	returns	0.800
1 x≷y −	returns	0.200

Even with 5% defects in the batch the odds are about 1 out of 3 that no defect at all is detected in such a relatively large sample. And note that such sample tests are certainly not adequate for controlling industrial processes with overall defect probabilities less than 1%.

STAT encompasses many more statistical functions (e.g. *covariances*, means and standard deviations for weighted data, *geometric means* and *scattering factors*) – just look them up there and check the respective entries in the *IOI*.

You will find all accumulated sums of your data in Σ. Summon these sums individually by calling their names (no need to memorize any register numbers in this matter).

More examples of statistical applications can be found in the manuals of various vintage *HP* calculators, especially the *HP-27* and *HP-21S*.

We strongly recommend you consult a good statistics textbook for more information about statistical methods in general, the terminology used, and the mathematical models provided, before applying them.

Real Numbers: Summary of Functions

The majority of the functions your *WP 43S* features are for calculations operating on *reals*. It provides many more than the numeric functions shown on pp. 20ff, 29ff, and 82ff in various applications and examples. See all *real* functions listed below:

- General mathematics:
  - *Monadic* functions:

    +/-, 1/x, x!, √x and x^2, ∛x and x^3, 2^x and lb x, 10^x and lg, e^x and ln, sin, cos, tan, and their inverses work as demonstrated above and you learned in school (see also pp. 125ff for more information about angular I/O),

    e^x-1 and ln(1+x) return more accurate results for $x \approx 0$,

    ceil returns the smallest integer $\geq x$, while floor returns the greatest integer $\leq x$,

    SDL n shifts digits left by n decimal positions, equivalent to multiplying x times 10^n, SDR ***n*** shifts digits right by ***n*** decimal positions, equivalent to dividing ***x*** by 10^n,

for sinh, cosh, tanh, and their inverses cf. pp. 94f,

$(-1)^x$ returns $cos(\pi x)$ for non-integer ***x***.

- *Dyadic* functions:

  [+], [−], [×], [/], [y^x], and $\sqrt[x]{y}$ work as was shown above and you learned in school; use …

  IDIV for integer division

  (e.g. **7.8** [ENTER↑] **3.2** [INTS] IDIV returns 2)

  (and IDIVR if you want also the remainder returned in **Y**),

  $log_x y$ for the logarithm of ***y*** for the base ***x***

  (e.g. **625** [ENTER↑] **5** [EXP] $log_x y$ returns 4),

  [RMD] for the remainder of ***y/x*** (see p. 143 for examples),

  MOD for ***y*** mod ***x*** (see p. 144 for examples),

  max (or min) for the maximum (or minimum) of ***x*** and ***y***; and

  || returns $\left(\frac{1}{x}+\frac{1}{y}\right)^{-1}$ for $x \times y \neq 0$ and 0 else, being handy in electrical engineering in particular.

- *Triadic* functions:

  ×MOD returns $(z \cdot y) \bmod x$ for $x > 1$, $y > 0$, $z > 0$, and

  ^MOD returns $(z^y) \bmod x$ for $x > 1$, $y > 0$, $z > 0$

  (e.g. **73** [ENTER↑] **55** [ENTER↑] **31** [INTS] ^MOD returns 26).

- Isolating parts of numbers: Use…

  EXPT for the exponent of ***x*** and MANT for its mantissa,

  FP (or IP) for the fractional (or integer) part of ***x***,

  [|x|] for the absolute value of ***x***, and

SIGN for the *signum* of x; thus, SIGN returns 1 for $x > 0$, –1 for $x < 0$, and 0 for $x = 0$ or non-numeric data.

- Rounding:

  RDP ***n*** rounds x to n decimal places in FIX format (e.g. **1.234 567 89E−95** RDP **99** will return 1.2346×10^{-95}),

  ROUND rounds x using the current display format (like RND did on *HP-42S*),

  ROUNDI rounds x to next integer (½ rounds to 1), and

  RSD ***n*** rounds x to n significant digits.

- Conversions:

  →P converts rectangular *coordinates* to polar ones (cf. pp. 20f), while **R←** converts vice versa.

  Angular, *time*, and *date conversions* are covered on pp. 125ff and 189ff.

  For *unit conversions* see pp. 276ff.

- Boole's algebra:

  AND, **NAND**, **OR**, **NOR**, **XOR**, **XNOR**, and **NOT** operate on *reals* like these operations did in the *HP-28S*, i.e. x and y are interpreted before executing the operation. Zero is 'false' (= 0); any other number is 'true' (= 1).

  Example: **13.5** **ENTER↑** **–7.2** **BITS** **AND** returns 1.

- Probability and statistics (unless introduced and explained on pp. 96ff already):

  Γ(x) calculates the *Gamma function*,

  lnΓ returns the natural logarithm of the *Gamma function*, allowing also for calculating really great factorials:

  Example: What is *5432!* ?

Remember $\Gamma(x+1) = x!$ So, entering **5433** X.FN lnΓ **10** ln / returns 17 931**.**480 374 010 87 as decadic logarithm of the result. Calling PARTS FP 10^x will return some 3**.**023 553 598 420 006 for its mantissa. Thus, 5 432**!** ≈ 3**.**024 × $10^{17\,931}$.

RAN# returns a (pseudo) random *real* number between 0 and 1,

SEED stores a seed (i.e. a start value) for RAN#,

RANI# returns a (pseudo) random integer number $\in [x, y]$;

and the other contents of

PROB cover combinations, permutations, and the 14 distributions introduced on pp. 97ff.

Σ contains all accumulated sums of your data, callable by their names.

In STAT, you find the summation commands Σ+ and Σ-, various mean values ($\bar{x}$, $\bar{x}_w$, $\bar{x}_G$, $\bar{x}_H$, $\bar{x}_{RMS}$), sample standard deviations (s, s_w) and standard errors (s_m, s_{mw}), population standard deviations (σ, σ_w), various scattering factors (ε, ε_m, ε_p), as well as all commands related to curve fitting (L.R. etc.).

Turn to the *ReM* for comprehensive information about all the statistical and probability functions provided on your *WP 43S*.

- Percentages:

% calculates $^{xy}/_{100}$, leaving y unchanged (so you can easily calculate another percentage of the same base after CLX).[108]

Example (from the *HP27 OH*):

If you buy a new car, you have to figure the sales tax percentage, then add that to the purchase price to find the total cost of the car. ... For example, if the sales tax on a $6200 car is 5%, what is the amount of the tax and total cost of the car?

---

[108] Actually, that's the (almost only) real benefit of the function %.

6200 ENTER↑ **5** FIN % returns 310. US$ for the sales tax;
\+ returns 6 510. US$ for the total cost.

If the dealer gives you a 10% discount on the car, what will your total cost be?

6200 ENTER↑
10 % − returns 5 580. US$ for the discounted price;
5 % + returns 5 859. US$ for the total cost.

Δ% calculates the percentage of change from y to x, returning $100\frac{x-y}{y}$, leaving y unchanged (for same reason as with %). You can use Δ% also for calculating *markup*[109] or *margin*:[110]

Example:

You purchase ink cartridges for 21.99 US$ wholesale and retail them for 26.50 US$. What percent is your *markup* and what percent is your *margin*?

21.99 ENTER↑ **26.5** Δ% returns 20.5 % *markup*.
26.5 ENTER↑ **21.99** Δ% returns −17.0, i.e. 17 % *margin*.

%MRR calculates the mean rate of return in % per period, i.e. $100\left(\sqrt[z]{\frac{x}{y}}-1\right)$ with y = present value, x = future value after z periods,

%T calculates $100\ {}^{x}/_{y}$ (called "% of total"), leaving y unchanged,[111]

%Σ returns $100\ {}^{x}/_{\sum x}$, and

[109] *Markup* is the price difference as a percentage of cost (wholesale) price.

[110] *Margin* is the price difference as a percentage of selling (retail) price.

[111] I still wait for somebody convincing me of the use of this financial function. Well, it preserves y, but else? Please see also %+MG and the corresponding footnote.

%+MG calculates a sales price by adding a *margin*[110] of x % to the cost y; you may use %+MG for calculating net amounts as well – just enter a negative percentage in x.

Example:

Total billed = 221,82 €, VAT = 19%. What is the net?

221.82 ENTER↑ **19** +/- FIN %+MG returns 186,40.[112]

- Advanced mathematics (see the *ReM, App. H* for comprehensive information about the functions following):
  - *Monadic* functions:

    B_n and B_n^* return the *Bernoulli numbers*,

    erf and erfc the *error function* and its complement,

    FIB the extended *Fibonacci number*,

    g_d and g_d^{-1} the *Gudermann function* and its inverse, and

    NEXTP the next *prime* number greater than x ;

    sinc returns $sin(x) / x$ for $x \neq 0$ and 1 for $x = 0$ (input shall be supplied in *radians* – see pp. 125f),

    W_p returns the principal branch of *Lambert's* W for given $x \geq -1/e$, W_m the negative branch of it,

    W^{-1} returns x for given W_p (≥ −1), and

    ζ(x) *Riemann's Zeta function*.

---

[112] Every engineer or scientist will be able to produce the very same result significantly faster via **221.82** ENTER↑ **1.19** /.

Seeing functions like %+MG and %T in particular provided on financial calculators, however, you may get the impression that average financial people might be mathematically slightly challenged and need some extra support. On the other hand, there is a saying in technical quarters (before 2008 already): '*Looking at the results financial people produce with plus and minus alone, their access to more advanced operations should be strictly limited*' (originally: "*Wenn man sieht, was Kaufleute mit plus und minus alles anstellen, sollte man sie an höhere Rechenarten erst gar nicht ranlassen*").

Call H_n for the *Hermite polynomials* for probability and

H_{np} for the *Hermite polynomials* for physics,

L_n for *Laguerre's polynomials* and

$L_{n\alpha}$ for *Laguerre's generalized polynomials*,

P_n for the *Legendre polynomials*,

T_n for the *Chebyshev polynomials of 1st kind* and

U_n for the *Chebyshev polynomials of 2nd kind*.

- *Dyadic* functions:

  AGM returns the *arithmetic-geometric mean*,

  $J_y(x)$ the *Bessel function of 1st kind* and order y,

  $\beta(x,y)$ *Euler's Beta function*,

  lnβ the natural logarithm of *Euler's Beta function*,

  γ_{xy} the *lower incomplete gamma function*,

  Γ_{xy} the *upper incomplete gamma function*, and

  $I\Gamma_p$ and $I\Gamma_q$ return the *regularized gamma function* (1 of 2 kinds).

- *Triadic* function*:*

  I_{xyz} returns the *regularized beta function*.

Angles and Trigonometric Functions

For dealing with *angles* on your *WP 43S*, you may choose out of five *angular display modes* (*ADM*) featured: DEG, RAD, GRAD, MULπ, and D.MS.[113] *Angles* are entered as *reals*. They are interpreted according to the current *ADM* as indicated in the *status bar* by ∡°, ∡ʳ, ∡ᵍ, ∡π, or ∡” (cf. p. 75) as soon as a function expecting angular input is called.

Exception: *Sexagesimal angles* must be entered in the format ddddd.mmsspp – with ddddd standing for integer *degrees*, mm for *angular minutes*, ss for *seconds*, and pp for hundredth of *seconds* – terminated by d.ms.

Example:

Entering **12.3454321** d.ms returns 12°34’54.32”.

There are some functions (e.g. ARCSIN) operating on *reals* and returning *angles*. The returned values will be automatically tagged according to the current *ADM*. Assume FIX 3 and RDX. set for the following **examples**:

In *ADM* ...	.5 TRI arccos will return …
∡ʳ	1.047ʳ
∡π	0.333π
∡°	60.000°
∡”	60° 0’ 0.00” [114]
∡ᵍ	66.667ᵍ

[113] All *ADM* setting commands except D.MS are found in MODE.

Translator’s note: The traditional calculator notations DEG and GRAD are misleading in German at least: DEGrees on your *WP 43S* mean “*Grad*”, while calculator GRADes are generally called “*Gon*” in Continental Europe.

[114] Note there are no leading zeroes in the angular *minutes* and *seconds* sections. And this *ADM* can neither take nor display anything smaller than 0.01”. On the other hand, it will display down to that fraction always and cannot be shortened.

Whenever you see a number formatted alike on your *WP 43S* you know it is an *angle*. – Other functions presume their inputs being *angles*, e.g. SIN. Decimal inputs are generally interpreted as *angles* of the current *ADM*.

14 angular conversions are provided, all found in [L→]:

From … / to …	*sexa-gesimal degrees*	*decimal degrees*	*radians*	*grades/ gon*	*multiples of π*	current *ADM* or tagging
sex. degrees	—	D→D.MS	—	—	—	→D.MS
dec. degrees	D.MS→D	—	R→D	—	—	→DEG
radians	—	D→R	—	—	—	→RAD
grades/gon	—	—	—	—	—	→GRAD
multipl. of π	—	—	—	—	—	→MULπ
current *ADM*	D.MS→	DEG→	RAD→	GRAD→	MULπ→	—

Example:

DISP FIX 5		
MODE MULπ		Choose *multiples of π* as *ADM* and ∡π will appear in the *status bar* and stay there for the time being.
300 1/x	0.003 33	So π/300 …
L→ →RAD	0.010 47 r	are 0.010 47 *radians*
→DEG	0.600 00°	or exactly 0.6°
→D.MS	0°36' 0.00"	or 36 *angular minutes*
→MULπ	0.003 33π	equivalent to π/300 still.

Note →RAD 'knew' it had to convert from *multiples of π* since this function expects angular input and took the current *ADM* setting into account. Angular output of operations is tagged and will stay so. Thus, →DEG above converted from *radians*, →D.MS from *decimal* and →MULπ from *sexagesimal degrees* since the respective inputs were tagged.

You have learned about trigonometric functions in school. Thus, we just demonstrate their operation on *angles* with one example.

Example (found in the *HP-25 OH*):

Lovesick sailor *Oscar Odysseus* dwells on the island of *Tristan da Cunha* (37°03'S, 12°l8'W), and his sweetheart, *Penelope*, lives on the nearest island. Unfortunately for the course of true love, however, *Tristan da Cunha* is the most isolated inhabited spot in the world. If *Penelope* lives on the island of *St. Helena* (15°55'S, 5°43'W), use the following formula to calculate the great circle distance that *Odysseus* must sail in order to court her. [115]

Solution:

The formula for the great circle distance ***d*** in *nautical miles* is:

$$d = 60 \times arccos[sin(B_s)sin(B_d) + cos(B_s)cos(B_d)cos(L_d - L_s)]$$

with B_s and L_s being the latitude and longitude of the start (*Tristan da Cunha*) and B_d and L_d being the latitude and longitude of the destination (*St. Helena*). [116] Hence, with the numbers inserted, this formula reads:

$$d = 60 \times arccos[sin(37°03'S)\ sin(15°55'S) + cos(37°03'S)\ cos(15°55'S) \times cos(5°43'W - 12°18'W)]$$

Set the appropriate number of decimals and calculate from inside out, remembering the trigonometric functions assume their input being in the current *ADM* as indicated in the *status bar*.

d.ms		Since we will use *sexagesimal degrees* throughout this calculation, we set *ADM* accordingly.
DISP FIX 2		We will not need more decimals displayed.
5.43 d.ms ENTER↑	5°43' 0.00"	
12.18 d.ms	12°18' 0.00"	
−	-7°15' 0.00"	

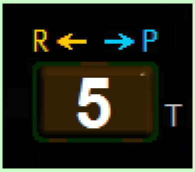
[115] This example was reprinted thereafter in each and every *HP* scientific pocket calculator manual until the *HP-41C/41CV OHPG*.

[116] This formula means that 1 nmi corresponds to 1 *angular minute* on a great circle. This doesn't hold exactly but precisely enough for practical sailing. See U→ for nmi. Translator's note: *Latitude* means "*geographische Breite*" in German. Hence ***B*** is used in the formula above.

TRI cos	0.99	
15.55 STO J cos	0.96	
×	0.96	
37.03 STO K cos	0.80	
×	0.76	
RCL K sin	0.60	
RCL J sin	0.27	
×	0.17	
+	0.93	
arccos	21°55'24.66"	
.d	21.92	convert to a *real number*.
60 × returning	1 315.41	nmi that *Odysseus* must sail to visit *Penelope*.

Mixed Calculations: Coordinate Transformations in 2D, Flight Directions, Courses over Ground, etc.

Two functions are provided for converting polar or rectangular coordinates in two dimensions. Input and output data are in *stack registers* **X** and **Y** here.



→P converts 2D Cartesian coordinates ***x*** and ***y*** to polar magnitude or radius ***r*** in **X** and angle ***ϑ*** in **Y**.

Example (assuming *startup default* settings):

Convert (***x***, ***y***) = (**6**, **4.5**) to polar. Two decimals shall do.

Solution:

DISP FIX 2

4.5 ENTER↑ 6 →P returns

ϑ =	36.87°
r =	7.50

i.e. a vector of magnitude 7.5 pointing up right from the origin with an angle of some 37° to the positive ***x***-axis.

[R←] does the reverse, it converts 2D polar magnitude or radius ***r*** in **X** and angle ***ϑ*** in **Y** to Cartesian coordinates ***x*** and ***y***. Both functions honour the *ADM* settings and tags as described in previous chapter.

Example (continued):

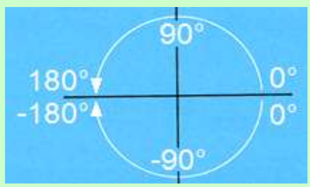
Convert the returned angle of the conversion executed above to *radians*, and then convert the resulting coordinates (***r***, ***ϑ***) to rectangular.

Solution:

[x≷y]	7.50 36.87°
[⮡] →RAD	7.50 0.64ʳ
[x≷y] [R←] returns	y = 4.50 x = 6.00

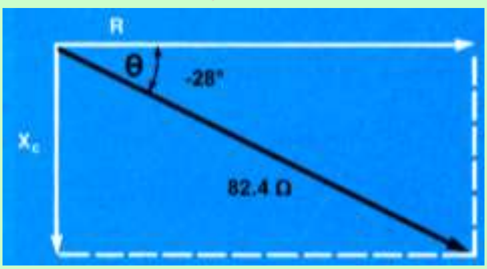
as expected.

Note angular input can range from −∞ to +∞; angular output, however, is confined to −180° to +180° or its equivalents, i.e. −π to +π in *radians*, −200g to +200g in *grades*, and −1 to +1 in *multiples of π*.



Example (triggered by the *HP-67 OHPG*)**:**

In an electronic circuit designed for alternating current, an overall impedance of 82.4 Ω is measured, and voltage lags current by 28°. Replacing said circuit by an equivalent containing just a resistor and a capacitor in series, what would be the resistance ***R*** and the capacitive reactance ***X***



Solution:

The values measured correspond to an impedance vector of magnitude

82.4 pointing down right at an angle of −28° to the positive ***x***-axis. ***R*** is its component parallel to the ***x***-axis, and X_C is its perpendicular component parallel to the ***y***-axis:

DISP FIX 0 1

−28 ENTER↑ 82.4

R← returns

y = -38.7
x = 72.8

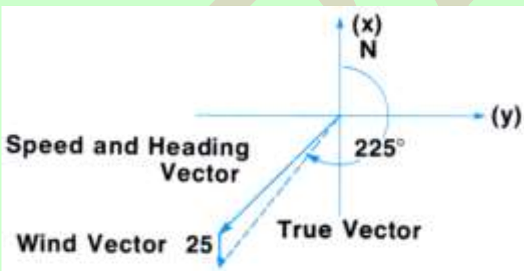
i.e. a resistance of 72.8 Ω and a reactance of 38.7 Ω.

By the way, you can use →P and R← also to convert 3D cylinder coordinates to Cartesian and vice versa, since ***z*** is kept unchanged.

Having learned about →P and R← as well as about Σ+ and Σ−, we can profit from combining these functions. Here is an example:

Example (from the *HP-25 OH*):

The instruments in fearless bush pilot *Apeneck Sweeney's* converted P-41 indicate an air speed of 125 *knots* and a heading of 225°. However the aircraft is also being buffeted by a steady 25-*knot* wind that is blowing from north to south. What is the actual course and speed of the aircraft?



Solution:

Combine the vector indicated on the aircraft instruments with the wind vector to yield the actual course and speed. Convert the vectors to rectangular, then combine the ***x***- and ***y***-coordinates in the statistical summation *registers*. Finally, recall the summed ***x***- and ***y***-coordinates and convert them to polar coordinates giving the actual vector of the aircraft. (North becomes the ***x***-coordinate in order that the problem corresponds with navigational convention.)

CLR CLΣ clears the summation *registers*.

DISP FIX 2

225 ENTER↑ **125** indicated air speed and heading.

R← returns

y = -88.39
x = -88.39

STAT Σ+ adds *x* and *y* to the summation *registers*.

180 ENTER↑ **25** north wind.

R← returns

y = 0.00
x = -25.00

Σ+ adds *x* and *y* to the summation *registers*.

SUM recalls the summation *registers* ***Σx*** and ***Σy***

→P returns

ϑ = -142.06°
r = 143.77

x≷y **360** + returns 217.94°

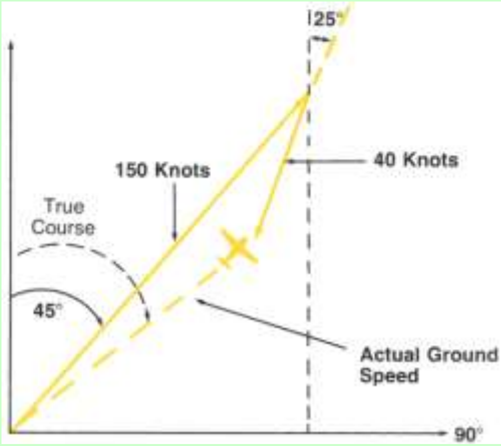
(we have to change the angle to become positive for being in line with navigational convention).

So, Mr. *Sweeney* is actually flying at 143.77 *knots* on a course of 217.94° over ground. Note we will demonstrate an alternative way for solving this kind of 2D vector problems on p. 158.

A similar example appeared first in the *HP-55 OH* and was copied then for some years. We quote the respective text from the *HP-33 OH*:

Example:

On his way to search for an albino caribou, grizzled bush pilot *Apeneck Sweeney's* converted *Swordfish* aircraft has a true air speed of 150 *knots* and an estimated heading of 45°. The *Swordfish* is also being buffeted by a headwind of 40 *knots* from a bearing of 25°. What is the actual ground speed and course of the *Swordfish*?



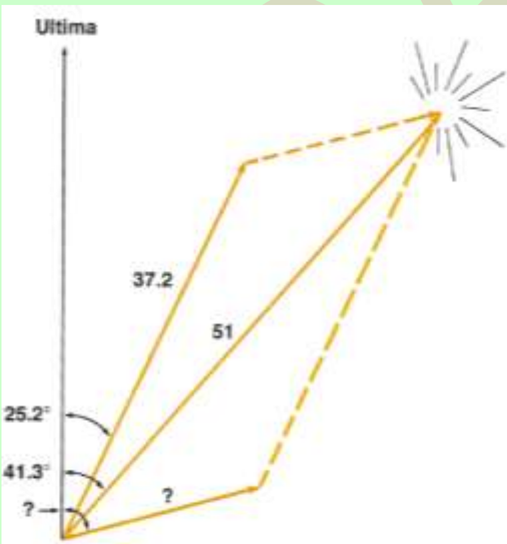
Start of solution:

Method 1: The course and ground speed are equal to the difference of the two vectors.

Method 2: Taking into account that a bearing of 25° equals a heading of 25° + 180° = 205°, the corresponding headwind vector may be added (cf. the *HP-97 OHPG*).

We leave it to you to solve this problem using [Σ+] and [Σ−] (but give you the results for crosschecking: 51.94° and 113.24 *knots*).

Additionally, here is an advanced problem from a universe far, far away:



Example from the *HP-32 OH*:[117]

Federation starship *Felicity* has emerged victorious from a furious battle with the starship *Θανατος*[118] from the renegade planet *Maldek*. However, its automatic pilot is kaput,[119] and its main thrust engine is locked on at 37.2 *meganewtons* (MN) directed along an angle of 25.2° from the star *Ultima* (= Latin for 'the last'). Consulting the ship's star map, the navigator reports a hyperspace entrance vector of 51 MN at an angle of 41.3° from *Ultima*.

[117] … of 1978. Note the first episode of *Star Wars* was launched in 1977.

[118] Translator's note: This is the ancient Greek word for 'death', pronounced like 'Tunnatoss' in English but like 'Thanatos' in Spanish, Italian, French, German, and Finnish, for example. Actually, they printed *Thanatos* in the English handbook.

[119] Oh, why can't the (American) English learn to speak … ummh … spell?

To what thrust and angle should the auxiliary engine be set, for *Felicity* to achieve alignment with the hyperspace entrance vector?

Solution:

The required thrust vector of the auxiliary engine is equal to the hyperspace entrance vector minus the thrust vector of the main engine. The vectors are converted to rectangular coordinates using R←, and their difference is calculated using Σ+ and Σ−. This difference is recalled to the **X**- and **Y**-*registers* using SUM. Then, these rectangular coordinates of the auxiliary engine thrust vector are converted to polar coordinates using →P.

CLR CLΣ clears the summation *registers*.

41.3 ENTER↑ **51** hyperspace entrance vector

R← returns

y = 33.66
x = 38.31

STAT Σ+ adds the *x* and *y* components of the hyperspace entrance vector to the summation *registers.*

25.2 ENTER↑ **37.2** main engine thrust vector

R← returns

y = 15.84
x = 33.66

Σ- subtracts the *x* and *y* components of the main engine thrust vector from the summation *registers.*

SUM recalls the summation *registers*:

Σy = 17.82
Σx = 4.65

→P returns

ϑ = 75.36°
r = 18.42

meaning the auxiliary engine shall be set at 18.42 MN and an angle of 75.36° from *Ultima*.[120]

[120] Those looking for an extra challenge can compute now how flat the crew of *Felicity* will become within *seconds* after the auxiliary engine is ignited.
By the way, the plane of action in 3D space seems to be defined sufficiently by *Felicity*, *Ultima*, and said hyperspace entrance (hopefully its center) here with all parameters specified to three digits maximum – a proper error calculation would have been

Real adepts of vector algebra may prefer subtracting the main engine thrust vector first and adding the hyperspace entrance vector second. This will work as well although the count of ‘data points’ will become negative once – simply don’t bother.

See the operating manuals of vintage *HP* calculators (especially the *HP-27*) for further applications from the areas of mathematics (e.g. triangle solutions), navigation, and surveying.

Angles: Summary of Functions

The number of functions operating on and with *angles* is quite limited. They are important nevertheless. See all functions listed below:

- General mathematics:
  - *Monadic* functions:

    sin, **cos**, and **tan** operate on *angles* and return *reals*,

    arcsin, **arccos**, and **arctan** operate on *reals* and return *angles*,

    [+/-] returns $x \times (-1)$ for closed input (a.k.a. ‘unary minus’).
  - *Dyadic* functions:

    [+], [−], [×], and [/] work as specified in the matrices on pp. 71f,

    max (or **min**) return the maximum (or minimum) of x and y.
- Rounding:

  ROUND rounds x using the current display format (cf. pp. 118f),
- Conversions:

  [→P] converts rectangular *coordinates* to polar ones (cf. pp. 20f), while [R←] converts vice versa. Cf. the examples on pp. 128ff.

  Angular conversions are covered comprehensively on p. 126.

---

appreciated. This problem was reprinted in the *HP-34C OH* one year later. It vanished in hyperspace thereafter, without a trace.

Integers: Input and Displaying

Any single number (e.g. a counted value) you enter without using [.] or [E] is regarded as an integer by your *WP 43S* (cf. pp. 68f). It allows for integer computing in fifteen bases from binary to hexadecimal.

Any single number displayed without any punctuation on your *WP 43S* is an integer (see examples below). And it will stay integer as long as it is exclusively combined with other integers and only integer functions operate on it; else it will be converted to another *data type* (cf. the matrices on pp. 71f and *Section 3* of the *ReM*). Note that any closed integer $\boldsymbol{x}$ will be converted to a *real number* by [.d], while even an open one will be converted to an *angle* by any angular conversion (cf. p. 74).

There are two kinds of integers provided on your *WP 43S*: integers of finite length (called *short*) and of almost arbitrary length (called *long*).

Long integers are useful e.g. for numeric tasks. If you enter a number of arbitrary length just without using [.] or [E], it is taken as a *long integer* of base 10. For **example**,

111 111 111 [x^2] returns 12 345 678 987 654 321

Note the number is adjusted to the right again when closed, though no point (or comma) is displayed. A 17-digit result is shown with ease. Large *long integers* ($>10^{21}$) will be displayed using the small font. Very large ones ($>10^{42}$) will be shown with an exponent instead of their least significant digits; nevertheless, all their digits are kept internally, so *long integers* can be of very high precision.

Example (a mathematical problem solved in 2019):

It was proven for integers $\boldsymbol{n}$ from 1 up to 100 that they[121] can be expressed as sum of three integer cubes $\boldsymbol{n} = \boldsymbol{i}^3 + \boldsymbol{j}^3 + \boldsymbol{k}^3$ – except for 42. In September 2019, two mathematicians of Bristol and Boston published that the numbers 12 602 123 297 335 631, 80 435 758 145 817 515, and −80 538 738 812 075 974 should solve this problem. Verify!

[121] Unless $\mathrm{mod}(\boldsymbol{n};9) = 4\ or\ 5$. See two chapters below for the function mod.

12 602 123 297 335 631 EXP x^3 SHOW returns

2 001 387 454 481 788 542 313 426 390 100 466 780 457 779 044 591

80 435 758 145 817 515 x^3 + SHOW returns

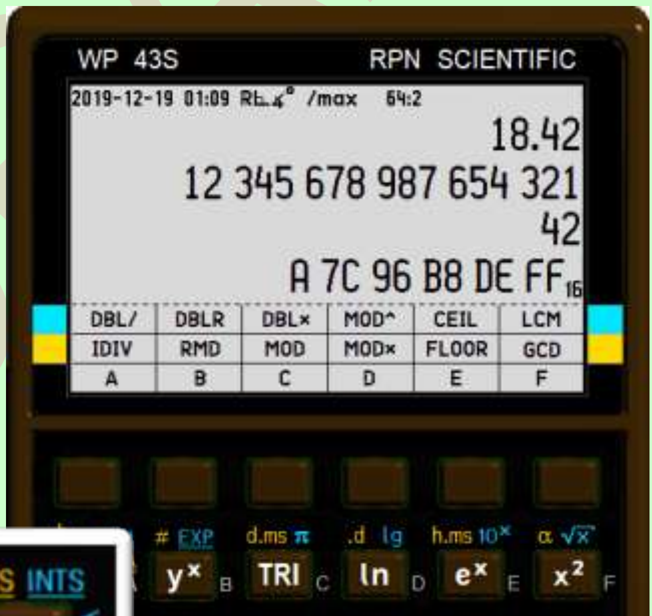
522 413 599 036 979 150 280 966 144 853 653 247 149 764 362 110 466

−80 538 738 812 075 974 x^3 + returns 42 **!**

This is all you need to know about entering and displaying *long integers* – turn to pp. 143ff for further information about calculating with them.

Short integers feature a finite *word* size (up to 64 *bits*) and are especially useful for computer logic and system design tasks incl. debugging. Your *WP 43S* encompasses all the integer and bit manipulation operations of the dedicated *Computer Scientist's HP-16C* and even all the bases and the entire extended function set of the *WP 34S*.

Short integers are entered with trailing # ***base*** (= 2 … 16). For decimal *short integers*, you may use # D instead of # 1 0, for hexadecimal # H instead of # 1 6. Open INTS (see its top *view* displayed here) for the digits **A** … **F** required for numeric input in bases > 10.



From the 2nd integer input on, you can save keystrokes: If you enter a new number omitting # and base (as well as ., E, and CC), your *WP 43S* takes it as a *short integer* of the same base you keyed in before – as long as you did not enter any other *data type* in between.[122]

Word size and *integer sign mode* (*ISM*) settings are indicated in the *status bar* using a format `ww:x`. Therein, `ww` denotes the *word* size in *bits* and `x` is 1 or 2 for *1's or 2's complement*, respectively, u for *unsigned*, or s for *sign-and-mantissa mode* (cf. p. 76); these *ISM's* control the handling of negative numbers (see examples below).

Carry and *Overflow* – if set – will be shown as c or o or o/c, respectively, trailing *ISM* display in the *status bar*. Both behave like they did on *HP-16C* or *WP 34S*, corresponding to *system flags* (cf. p. 55) – if you want to set, clear, or check them one by one, use the commands provided in FLAGS.

Example:

Enter FLAGS SF SYS.FL LEAD.0 (or FLAGS SF L)

INTS ▲ WSIZE 1 2	This allows seeing all bits at a glance easily.
147 ENTER↑	Enters 147 (base 10)
# 2	Converts decimal 147 to binary.
1COMPL	and you will see [123]

$0000\ 1001\ 0011_2$ and – after +/- – $1111\ 0110\ 1100_2$.

Obviously +/- in 1COMPL flips every bit, equivalent to NOT here.

Return to the original number via +/-, press 2COMPL, and you will get

$0000\ 1001\ 0011_2$ and – after +/- – $1111\ 0110\ 1101_2$.

Note the negative number equals the inverse + 1 in 2COMPL.

---

[122] This shortcut will be left as soon as you enter a ., E, CC, or # in input, even if deleted thereafter.

Illegal digits keyed in (e.g. **2** in base 2 or **B** in base 10) can be detected no earlier than said input is completed, so an error will be thrown then. You may key in more than the current *word* size can take – also this will be checked when input is closed.

[123] Note the gap automatically inserted every four bits here for easy reading this output.

Return via [+/-] again, press **SIGNMT** and you will see

$0000\ 1001\ 0011_2$ and – after [+/-] – $1000\ 1001\ 0011_2$.

Negating a number will just flip the top bit in SIGNMT (hence the name of this mode).

Return via [+/-] once more, press **UNSIGN** and you will get

$0000\ 1001\ 0011_2$ and – after [+/-] – $1111\ 0110\ 1101_2$.

Note the 2nd number looks like in 2COMPL, but in addition an *overflow* is set here – see the ° in the *status bar* trailing the *ISM*.[124] Thus, pressing [+/-] will not suffice anymore for returning to the original number here; you must clear the *overflow flag* explicitly by FLAGS **CF** **SYS.FL** **OVERFL**.

As you have seen, positive numbers stay unchanged in all those four modes. Negative *short integers*, on the other hand, are displayed in different ways. Therefore, taking a negative integer in one mode and switching to another one will lead to different interpretations.

Example:

The fixed bit pattern representing

-147_{10} in **12:2** will be displayed as…

-146_{10} in **12:1** , as…

$-1\ 901_{10}$ in **12:s** , and as…

$3\ 949_{10}$ in **12:u** . You can verify this easily.

Keeping the mode and changing bases will produce different views of the constant bit pattern as well.

---

[124] This needs explanation, since changing signs should have no meaning in unsigned mode per definition. Thus, [+/-] should be illegal here or result in no operation at least. "In unsigned mode, the most significant bit adds magnitude, not sign, so the largest value represented by a 12-*bit* word is 4095 instead of 2047" (quoted from the *HP-16C Computer Scientist Owner's Handbook* of April 1982, p. 30).

Unfortunately, however, [+/-] in unsigned mode was allowed by the designers of the *HP-16C* and implemented as shown above; so we follow that implementation for sake of backward compatibility, though frowning.

Example:

Compare the outputs for different bases in 12:2 :

	-147₁₀	corresponds to…
# 2	1111 0110 1101₂	,
# 3	-12 110₃	,
# 4	33 12 31₄	,
# 5	-1 042₅	,
# 6	-403₆	,
# 7	-300₇	,
# 8	75 55₈	, and
# 9	-173₉	.

You may have noticed that the displays for bases 2, 4, and 8 look similar, presenting all twelve bits to you, while in the other bases a signed mantissa is displayed instead. There are also different separator intervals; they are fixed for *short integers* unless GAP **0** is set by you. These different display formats (and more) take into account that bases 2, 4, 8, and 16 are most convenient for bit and byte manipulations and further close-to-hardware applications. The bases in between will probably gain most interest in dealing with different number representations and calculating therein, where base 10 is the common reference standard.[125]

Let's look to bigger *words* now:

Example (continued):

Enter FLAGS CF L

INTS ▲ WSIZE 6 4 UNSIGN

▼ 9 3 A 1 4 C 6 # H (cf. the *menu* shown on p. 136).

Then your *WP 43S* will display

9 3A 14 C6₁₆

[125] During numeric input, however, a gap is inserted every 3 digits as for real numbers – your *WP 43S* cannot know in advance what you have in mind.

In binary representation, this number will need 28 digits and would look like

1001 0011 1010 0001 0100 1100 0110_2.

Obviously, your *WP 43S* cannot display a binary number of this size this way in a single row (no pocket calculator can as far as we know). Look what it does instead – enter # 2 for converting *x* to binary and you will see:

1001 0011 1010 0001 0100 1100 0110_2

This binary number is displayed using the small font provided. If leading zeros were turned on via FLAGS SF L, all 64 bits would be displayed in one row making use of a minimal font:

0000 0000 0000 0000 0000 0000 0000 0000 0000 1001 0011 1010 0001 0100 1100 0110_2

… with the 36 most significant bits all containing 0.

Integers: Bitwise Operations on Short Integers

Your *WP 43S* carries all the bitwise operations you may know from the vintage *HP-16C Computer Scientist*, plus some more you may have learned with the *WP 34S*. You find them all in BITS. Generally, bits in a *word* are counted from right to left, starting with number 1 for the least significant bit. This convention is important for specifying correct bit numbers in the operations BC?, BS?, CB, FB, and SB.

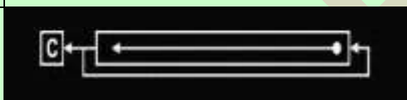
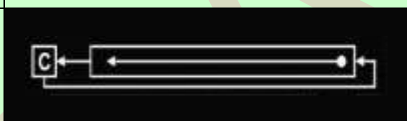
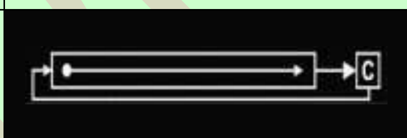
The following **examples** deal with 8-*bit words* showing leading zeros for easy reading.

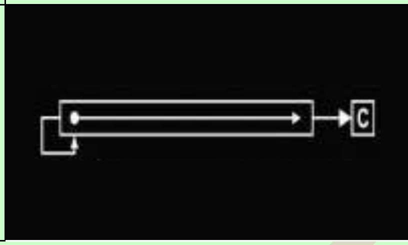
BITS ▲ WSIZE 8

FLAGS SF L

1011 0011 # 2 STO K

This is the common initial number for the operations presented in the table below. You find seven shift and rotate functions with schematic pictures herein like they were printed on the backplane of the *HP-16C*, wherein the boxed **C** represents the *carry* bit indicated in the *status bar* if set.

Operation	Schematic picture if applicable	E. g.	Output
Clear Bit		CB **5**	$1010\ 0011_2$
Flip Bit		FB **6**	$1001\ 0011_2$
Set Bit		SB **7**	$1111\ 0011_2$
Negate		NOT (¬)	$0100\ 1100_2$
Mirror		MIRROR	$1100\ 1101_2$
Rotate Left		RL **1** RL **2**	$0110\ 0111_2$ c $1100\ 1110_2$
Rotate Left through Carry		RLC **1** RLC **2**	$0110\ 0110_2$ c $1100\ 1101_2$
Rotate Right		RR **1** RR **2** RR **3**	$1101\ 1001_2$ c $1110\ 1100_2$ c $0111\ 0110_2$
Rotate Right through Carry		RRC **1** RRC **2** RRC **3**	$0101\ 1001_2$ c $1010\ 1100_2$ c $1101\ 0110_2$
Shift Left		SL **1** SL **2**	$0110\ 0110_2$ c $1100\ 1100_2$
Shift Right		SR **1** SR **2**	$0101\ 1001_2$ c $0010\ 1100_2$ c

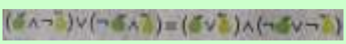
Operation	Schematic picture if applicable	E. g.	Output
Arithmetic Shift Right		ASR **3**	in 1/ 2COMPL: 1111 0110$_2$ in UNSIGN:[126] 0001 0110$_2$ in SIGNMT: 1000 0110$_2$

Now let's also look at the bitwise *dyadic* functions. We will continue using 8-*bit words* displayed as above for the following **examples:**[127]

Common input	**Y**	0110 1011$_2$
	X	1011 1001$_2$
Operation	**Symbol**	**Output**
AND	∧	0010 1001$_2$
NAND	$\bar{\wedge}$	1101 0110$_2$
OR	∨	1111 1011$_2$
NOR	$\bar{\vee}$	0000 0100$_2$
XOR	⊻	1101 0010$_2$
XNOR		0010 1101$_2$

See the *IOI* for these and further commands operating on bit level on integers (LJ and RJ, MASKL and MASKR, #B, and the tests BS? and

[126] The picture for ASR correctly describes this operation for *1's* and *2's complement modes* only. In all modes of the *HP-16C*, however, ASR **3** equals a signed division by 2^3, hence the different results for the latter two modes shown above. The other bitwise operations are insensitive to *ISM* setting. Turn to the *IOI* for further details.

[127] Remember: 

BC?). Most of them are found in BITS.

Finally, note that no such operation will set an *Overflow*. *Carry* is only settable by shift or rotate functions as demonstrated above. And ASR is the only bitwise operation being sensitive to *ISM* – ASR is the link to integer arithmetic operations.

Integers: Arithmetic Operations

Of the four basic arithmetic operations (+, −, ×, and /), the first three work with both kinds of integers as they do with *reals*; the only difference lies in precision: up to 64 digits precision for *short integers* in binary representation on your *WP 43S* or even (almost) infinite precision for *data type* 1. Take [+/−] as a multiplication times −1, and y^x as repeated multiplication. Depending on input parameters and mode settings, the OVERFLOW or CARRY *flags* may be set in such an operation (see pp. 146ff).

Divisions, however, must be handled differently in integer domain since the result cannot feature a fractional part here. Generally, the formula

$$\frac{a}{b} = (a \operatorname{div} b) + \frac{1}{b} \times \operatorname{rmd}(a; b)$$

applies; therein, the horizontal bar denotes *real* division, div represents integer division, and rmd stands for the remainder of the latter. While remainders for positive parameters are simply found, remainders for negative dividends or divisors may lead to confusion sometimes. The formula above, however, is easily employed for calculating such remainders (also for *reals* – see the first row of the **examples** here):

$\frac{25}{7} = 3 + \frac{1}{7} \times 4$ (and for a *real* case: $\frac{25}{7.5} = 3 + \frac{1}{7.5} \times 2.5$)

$\frac{-25}{7} = -3 + \frac{1}{7} \times (-4)$ $\Rightarrow$ $\operatorname{rmd}(-25; 7) = -4$

$\frac{25}{-7} = -3 + \frac{1}{-7} \times 4$ $\Rightarrow$ $\operatorname{rmd}(25; -7) = 4$

$\frac{-25}{-7} = 3 + \frac{1}{-7} \times (-4)$ $\Rightarrow$ $\operatorname{rmd}(-25; -7) = -4$

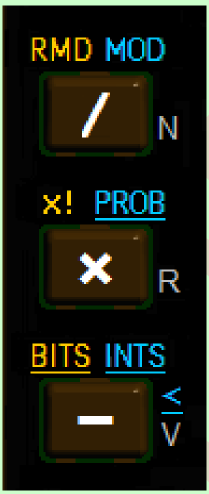
In general, $\mathrm{rmd}(a;b) \coloneqq a - b \times (a \operatorname{div} b)$ applies.

Unfortunately, there is a second function doing almost the same: it is called mod. With the same pairs of numbers as above, it returns:

$$\mathrm{mod}(25;7) = 4\ ,$$
$$\mathrm{mod}(-25;7) = 3\ ,$$
$$\mathrm{mod}(25;-7) = -3\ ,$$
$$\mathrm{mod}(-25;-7) = -4\ .$$

So mod (i.e. modulo) returns the same as rmd only if both parameters have equal signs. The general formula for mod is a bit more sophisticated than the one above:

$$\mathrm{mod}(a;b) \coloneqq a - b \times \mathrm{floor}\left(\frac{a}{b}\right)$$ with e.g. $\mathrm{floor}\left(\frac{25}{7}\right) = 3$ and $\mathrm{floor}\left(-\frac{25}{7}\right) = -4$.



By the way, this formula applies to *reals* as well. So it may be used straightforwardly for calculating e.g.

$$\mathrm{mod}(25.3;-7.5) = 25.3 - (-7.5) \cdot (-4) = -4.7\ .$$

These four functions are called IDIV, RMD, MOD, and FLOOR [128] on your *WP 43S* for obvious reasons. They are found in INTS (cf. p. 136), together with more integer operations like CEIL, ×MOD, and ^MOD (see p. 149 for an example); RMD and MOD are also on the keyboard as shifted function of /.

Furthermore, many exponential and logarithmic operations, x^2 and $\sqrt{x}$, x^3 and $\sqrt[3]{x}$, $x!$, COMB and PERM, as well as SIN, COS, and TAN operate on integers, too. Note that some of these functions will stay in integer domain while others may or will return *real* or even *complex numbers*. See the summary on pp. 148f for further information.

---

[128] Note FLOOR and CEIL are functions operating on a *real number* and returning a *long integer*.

Integers: Overflow and Carry with Short Integers

There are conditions where OVERFL and/or CARRY will be touched in arithmetic operations on *short integers* on your *WP 43S*. Note there is a maximum and a minimum integer displayable for each *word* size and *ISM* setting – let's call them I_{max} and I_{min}.

Example:

For four-*bit words* (i.e. WSIZE **4**), we get

- $I_{max} = 15$ and $I_{min} = 0$ for 4:u, while
- $I_{max} = 7$ and $I_{min} = -8$ for 4:2,
- $I_{max} = 7$ and $I_{min} = -7$ for 4:1 and 4:s.

Let's start from 1 incrementing by 1 and see what will happen in these various modes. And whenever OVERFLOW and/or CARRY will be lit in the *status bar* in this course, we will clear them (using
FLAGS CF SYS.FL OVERFL or FLAGS CF O and/or
FLAGS CF SYS.FL CARRY or FLAGS CF C)
before next increment:

4:u		4:2		4:1		4:s	
0001_2	1	0001_2	1	0001_2	1	0001_2	1
0010_2	2	0010_2	2	0010_2	2	0010_2	2
…	…	…	…	…	…	…	…
0111_2	7	0111_2	7	0111_2	7	0111_2	7
1000_2	8	1000_2 o	-8	1000_2 o	-7	1000_2 o c	-0
1001_2	9	1001_2	-7	1001_2	-6	0001_2	1
…	…	…	…	…	…		
1110_2	14	1110_2	-2	1110_2	-1		
1111_2	15	1111_2	-1	1111_2	-0		
0000_2 o c	0	0000_2 c	0	0001_2 c	1		
0001_2	1	0001_2	1				

For comparison, we start another turn from 1 following the same rules but <u>de</u>crementing by 1 instead:

4:u		4:2		4:1		4:s	
0001_2	1	0001_2	1	0001_2	1	0001_2	1
0000_2	0	0000_2	0	0000_2	0	0000_2	0
1111_2 o c	15	1111_2 c	-1	1110_2 c	-1	1001_2 c	-1
1110_2	14	1110_2	-2	1101_2	-2	1010_2	-2
…	…	…	…	…	…	…	…
1001_2	9	1001_2	-7	1000_2	-7	1111_2	-7
1000_2	8	1000_2	-8	0111_2 o	7	1000_2 o	-0
0111_2	7	0111_2 o	7	0110_2	6	1001_2	-1
0110_2	6	0110_2	6	…	…		
…	…	…	…	…	…		
0010_2	2	0010_2	2	0001_2	1		
0001_2	1	0001_2	1				

The most significant bit is #3 in 4:s and #4 in all other modes here.

With these results, I_{max}, and I_{min}, the general rules for setting and clearing CARRY and OVERFL in *ISMs* are as presented in the table overleaf:

Operation	**Effect on CARRY**	**Effect on OVERFL**
Shift and rotate	As demonstrated on pp. 141f.	None.
Boole's, MIRROR	None (cf. pp. 141f).	None.
|x|, ABS	None.	Clears OVERFL (but sets it for $x = I_{min}$ in 2COMPL).

Operation	Effect on CARRY	Effect on OVERFL
+, RCL+, STO+, INC, etc.	Sets CARRY if there is a *carry* out of the most significant bit, else clears CARRY.	Sets OVERFL if the result exceeds $[I_{min}; I_{max}]$, else clears OVERFL.
−, RCL−, STO−, DEC, etc.	Sets CARRY in a subtraction $\boldsymbol{m} - \boldsymbol{s}$ • in 1COMPL or 2COMPL if the binary subtraction causes a *borrow*[129] into the most significant bit, • in UNSIGN if $\boldsymbol{m} < \boldsymbol{s}$, • in SIGNMT if $\boldsymbol{m} < \boldsymbol{s}$ & $\boldsymbol{m} \cdot \boldsymbol{s} > 0$ Else clears CARRY.	Sets OVERFL if the result exceeds $[I_{min}; I_{max}]$, else clears OVERFL.
×, RCL×, STO×, +/−, $(-1)^x$, x^2, x^3, LCM, $x!$, etc.	None.	Thus, in UNSIGN, +/− always sets OVERFL and $(-1)^x$ does so for odd x.
2^x	Clears CARRY. Sets CARRY only if $x = -1$, or in UNSIGN if $x = \boldsymbol{wsize}$ or in the other modes if $x = \boldsymbol{wsize} - 1$	
y^x, 10^x	Sets CARRY for $x < 0$ (as well as for 0^0), else clears CARRY.	Sets OVERFL if the result exceeds $[I_{min}; I_{max}]$, else clears OVERFL.
e^x	Sets CARRY for $x \neq 0$, else clears.	
DBL×	None.	Clears OVERFL.

129 See the examples on previous page.

Translator's note: The so-called *borrow* in subtraction seems to be a specialty of the USA. See the subtle methodic differences in manual subtracting explained in *http://de.wikipedia.org/wiki/Subtraktion#Schriftliche_Subtraktion* . The corresponding English article is less instructive. Both *carry* and *borrow* translate to *Übertrag* in German.

Operation	Effect on CARRY	Effect on OVERFL
/ , RCL / , STO /, DBL /, LN, LOG_{10}, LOG_2, LOG_xy, $\sqrt{x}$, $\sqrt[3]{x}$, $\sqrt[x]{y}$	Sets CARRY if the remainder is ≠ 0, else clears CARRY.	Clears OVERFL but sets it in 2COMPL for the division $I_{min}/(-1)$.

Integers: Summary of Functions

Many of the numeric functions operating on *reals* also work for integers. In addition, there are some specialties as shown in the preceding chapters, and beyond:

- General mathematics:
  - *Monadic* functions:

    √x, ∛x, lb x, and lg return *long integers* if possible (else *real* or *complex numbers*) for *long integer* input [130] or just the *short integer* part of the solution for *short integer* input,

    |x|, x², x³, 2ˣ, and 10ˣ return integers as you expect, and

    +/– works for *short integers* as demonstrated on p. 137.

  - *Dyadic functions:*

    +, −, ×, and yˣ return integers as you expect,

    / returns an integer or a *real* in analogy to √x,

    IDIV returns just the integer part of the division always,

    RMD returns the remainder of y/x (cf. pp. 143f for examples),

    IDIVR combines IDIV and RMD,

[130] E.g. √x returns 8 for an input of 64, i.e. for a proper square, and 8.062… for an input of 65, for instance. For an input of −64, it may return 0.+i×8 (see the chapters about *Complex Numbers* below). In analogy, ∛x returns 2 for an input of 8, lb x returns 10 for an input of 1024, lg returns 3 for an input of 1000, etc.

MOD returns $y \bmod x$ (cf. p. 144 for examples),

$\sqrt[x]{y}$ and $\log_x y$ return integers or *reals* in analogy to $\sqrt{x}$
(e.g. **625** ENTER↑ **5** EXP $\log_x y$ returns 4),

max (or min) return the maximum (or minimum) of x and y,

GCD the *Greatest Common Divisor* of x and y and

LCM the *Least Common Multiple* (remember school?).

- *Triadic* functions:

  ×MOD returns $(z \cdot y) \bmod x$ for $x > 1$, $y > 0$, $z > 0$, and

  ^MOD returns $(z^y) \bmod x$ for $x > 1$, $y > 0$, $z > 0$
  (e.g. **73** ENTER↑ **55** ENTER↑ **31** INTS ^MOD returns 26).

- *Boole's* algebra:

  AND , NAND , OR , NOR , XOR , XNOR , and NOT operate bitwise on *short integers* as shown on p. 142. They operate on *long integers* like in the *HP-28S*, i.e. x and y are interpreted before executing the operation; zero is 'false' (= 0); any other number is 'true' (= 1), cf. p. 120.

- Bitwise operations are exclusively for *short integers*:

  CB , FB , SB , ASR , SL , SR , RL , RLC , RR , RRC , and MIRROR work as demonstrated on pp. 140ff.

  LJ (or RJ) justifies the bit pattern to the left (or right) within its *word* size,

  MASKL and MASKR create mask *words*,

  BC? and BS? test if the specified bit is clear or set,

  #B counts the number of bits set in x .

  See the *IOI* for more information about these commands.

- Probability (cf. pp. 96f):

  x! returns the factorial,

  C_{yx} calculates the number of *combinations* and

`Pyx` the number of *permutations*, while

`RANI#` returns a (pseudo) random integer number $\in [x, y]$.

- Advanced mathematics (see the *ReM*, *App. H* for more information):

  `Bn` and `Bn*` return the *Bernoulli numbers*,

  `FIB` the *Fibonacci number*, and

  `NEXTP` the next *prime* number greater than x .

Many more functions accept integer input but will return different, mostly *real* output. See the *IOI* and *Section 3* of the *ReM* for details.

Rational Numbers (Fractions)

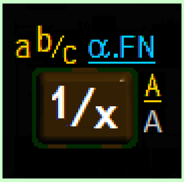
Fractions are handled like in previous *RPN* calculators. In particular, DENMAX sets the maximum allowable denominator (up to 9999, see the *IOP*). On your *WP 43S*, you can work with fractions like on the *HP-32SII* and its successors but with higher precision.

A fraction is **entered** directly by keying in a 2nd radix mark in numeric input (see the examples below). Here, the 1st radix mark is interpreted as a blank space, the 2nd as a fraction mark. This way of input is straightforward and logically coherent:

Examples:

Key in:	… and get in *startup default* format	
[1][2] [.] [3] [.] [4] [ENTER↑]	$12\ {}^{3}/_{4}$ =	
[1] [.] [2] [ENTER↑]	1.2	(decimal input)
[.] [1] [.] [2] [ENTER↑]	${}^{1}/_{2}$ =	
[.] [1][2] [ENTER↑]	0.12	(decimal input)
[1] [.] [.] [2] [ENTER↑]	$1\ {}^{0}/_{1}$ =	(= 1 ${}^{0}/_{2}$) [131]

Each closed *real number* on the *stack* will be **displayed** as a fraction after [a b/c] is pressed, after a fraction is entered, or after that number is combined with a fraction by an arithmetic operation. If the fraction displayed is exactly equal, slightly less, or slightly greater than the underlying *real number*, =, <, or > will trail this fraction display, respectively (see examples overleaf).



---

[131] This display of a pure integer number tells you unambiguously your *WP 43S* is in *proper fraction display mode*. In *improper fraction display mode*, ${}^{1}/_{1}$ = will be displayed instead. For comparison, note the *HP-32SII* reads [1] [.] [.] [2] as ½ – though this is not coherent with its other input interpretations (and does not even save keystrokes but adds confusion only).

Vice versa, each closed number ***x*** displayed as a fraction will be shown as a decimal *real number* after .d or after DISP ALL, FIX, SCI, or ENG. And a closed fraction ***x*** will be decomposed to its integer numerator in **Y** and its integer denominator in **X** by PARTS DECOMP.

There are two fraction display modes: *proper* and *improper fractions.*[132] a b/c toggles them. They are illustrated below. On your *WP 43S*, fraction display can handle numbers with absolute values greater than 10^{-4}; maximum denominator is 9 999 (greater denominators may be entered but will be reduced as soon as input is closed).

The following example comprises most aspects of fraction display:

Example (with *startup default* settings):

Enter:	and you will see:
3 1/x	3.333 333 333 333 333×10⁻¹
a b/c a b/c	1/3 >
	since 1/3 > 0.333 333 333 333 333 3.
1/x	3/1 =
	since this is exact.
78.40625 −	-2 413/32 =
x^2	5 822 569/1 024 =
2 ×	5 822 569/512 =

Now, press a b/c for converting this *improper fraction* to a *proper* one.[133] You will get

	11 372 105/512 =
11 /	1 033 4 713/5 632 <

This fraction is less than the *real* value, deviating less than 0.5/5 632 from it.

---

[132] Translator's note for German readers: *Proper fractions* decken sowohl *echte Brüche* (wie ¾) als auch *gemischte Brüche* (wie 2 ½) ab. Bei *improper f.* wird der ganzzahlige Anteil nicht herausgezogen, so dass hier der Zähler größer als der Nenner sein kann.

[133] This conversion was newly introduced on *RPN* calculators with the *WP 34S*.

Now, let's reduce the maximum denominator by

64 MODE DENMAX	64
R↓	1 033 41/49 <
FLAGS CF SYS.FL DENANY	
CF SYS.FL DENFIX	1 033 27/32 >

since DENANY and DENFIX both cleared allow for denominators being factors of DENMAX only (i.e. 2, 4, 8, 16, 32, and 64 here). This last fraction is greater than the real value; the fraction shown deviates from it by 0.5/32 maximum (and by 0.5/64 minimum – else the display would read 1033 53/64 instead).

Before closing this chapter about numbers displayed as fractions, we will not forget those isolated irrational islands in the vast sea of *SI* where you may come across dimensions like in the following **example:**

A calculator stand is specified to measure 9" × 3 1/2" × 5/8". It goes without saying that your *WP 43S* will support you also in such harsh environments. Only absolute greenhorns, however, will expect that a tight thin-walled box around this stand will displace

9 ENTER 3 . 1 . 2 ×	31 1/2 =
. 5 . 8 ×	19 11/16 =

cubic inches of water.

Instead, a magic conversion factor from *cubic inches* to so called *fluid ounces* is required now.[134] And this factor even depends on the country you are in! Though do not despair: in *Section 5* you will learn how to do this magic using your *WP 43S* – it takes just a little more time and effort than calculating with rational units.

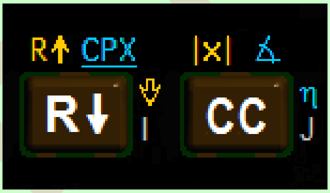
[134] Honestly, unless you grew up in such a place we bet you have assumed *fluid ounces* being a unit of mass, haven't you? Since you have heard of *ounces* once before and just thought … terribly wrong! Do not think there – you may run into deep troubles easily (though thinking less you might achieve top positions in administration – see recent experimental evidence).

Complex Numbers: Introduction

So far, we dealt with *reals* only (rational and integer numbers are mere subsets of *reals*). Your *WP 43S* can do more for you. Mathematicians know of more complex items than *reals*; these are called *complex numbers*. If you do not know of them, leave them aside; you can profit from your *WP 43S* perfectly without them.

If you know of *complex numbers*, however, note your *WP 43S* supports many operations in *complex* domain as well as it does in *real* domain.

Complex numbers may be **entered** using CC (see p. 307). With *startup default* settings, CC separates and concatenates *real* and *imaginary* part in numeric input.



Examples (with *startup default* settings):

3 + ***i*** × 4 is keyed in 3 CC 4 ENTER↑ while the display (set e.g. to FIX 5) shows in lowest numeric row:

3

3 +i×

3 +i×4

3.000 00 +i×4.000 00

You enter the *real* part first – CC closes it – the *imaginary* part second as you write the number.[135]

Input of negative *complex numbers* works in full analogy to *real number* input (cf. p. 25). Following our example above,

[135] Entering both parts vice versa would be more like *RPN*: first the imaginary part, then CC interpreted as ***i*** ×, finally the *real* part to be added. But it was decided differently for the *HP-42S* already. So we follow tradition here.

For those of you working on the field of electronic engineering, an alternate format is provided employing the letter j for the *complex unit* (the respective is called CPXj for obvious reasons).

3 – ***i*** × 4	is keyed in	3 CC 4 +/- ENTER↑,
–3 + ***i*** × 4	is keyed in	3 +/- CC 4 ENTER↑,
–31 – ***i*** × 42	is keyed in	3 1 +/- CC 4 2 +/- ENTER↑.
	Alternatively, use	3 1 CC 4 2 ENTER↑ +/- here.

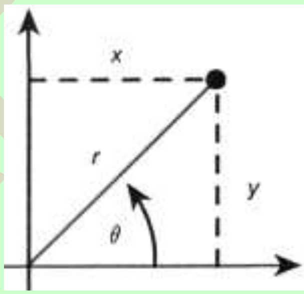
Choosing scientific notation, e.g. SCI 5, this last number will be displayed like

-3.100 00×10¹ -i×4.200 00×10¹

Depending on display format set, this may be shown more compact (allowing for one decimal more):

-3,100 000·10¹ -i·4,200 000·10¹

Alternatively to rectangular notation, *complex numbers* may be written in polar notation as well. With polar notation set (by FLAGS SF SYS.FL POLAR or FLAGS SF X causing ⊙ lit in the *status bar*), its *magnitude* (or *radius*) ***r*** shall be entered first for a new *complex number* and its *phase* (angle or *argument*) ***ϑ*** second. This ***ϑ*** may be entered in any angular notation; though often *radians* or *multiples of π* make most sense here (e.g. set MODE RAD causing ∡ʳ lit in the *status bar*).



Example:

With *polar display mode*, *radians*, and FIX 2 set, the *complex number* (5 ; 1.2ʳ) is keyed in **5** CC **1.2** ENTER↑ with the display showing in lowest numeric row successively:

5

5 ∡

5 ∡ 1.2

5.00 ∡ 1.20ʳ

Special cases: If a negative magnitude is entered, it is made positive and ***ϑ*** is increased by π and then normalized (i.e. ***ϑ*** will never exceed the interval (−π, π] in *radians* or its equivalents – cf. p. 129). Larger phase input is legal, but the output will be normalized always. If **0** is entered for the magnitude, *ϑ* will be set to **0** as well.

Composing and decomposing a *complex number*: Alternatively to entering a *complex number* directly, it may be generated from two closed *reals* provided in **X** and **Y**. If ⊾ is lit, CC will take *y* as *real* part and *x* as *imaginary* part composing the *complex number*. If ⊙ is lit on the other hand, it will take *y* as magnitude and *x* as phase of the new *complex number* (compare numeric input above).

Example:

MODE CF X CF I DEG	These three entries return to *startup default* settings R⊾∡° .
4 ENTER↑ −3 EXIT	4. -3.
	EXIT closes input without disturbing the stack.
CC composes a *complex number* out of *x* and *y* now, lights ℂ and returns[136]	4.-i×3.
SF X	turns ⊾ to ⊙
and displays	5.∡-36.869 897 645 844 02°

Vice versa, CC may also cut a *complex number* *x* into two *reals* in **X** and **Y** following the same rules.

Example (continued):

CC returns	r = 5. ϑ = -36.869 897 645 844 02°

[136] Whenever a *complex number* is returned, your *WP 43S* will set CPXRES and ℂ will be lit in the status bar unless set before.

C⊙ remains lit in the *status bar.*

RAD CC	returns	5.∡-6.435 011 087 933×10⁻¹ʳ
CF X	turns ⊙ to ⊾ and displays	4.-i×3.
CC	returns	Re = 4. Im = -3.

C⊾∡ʳ remains lit in the *status bar.*

Generally, *complex number* outputs follow *real number* formats (see pp. 80ff). The number of displayable decimals, however, may be limited by screen space. If you want to view both parts of a *complex number* in higher precision, press CC, watch, and press CC again.[137]

***Complex* results in calculations:** As long as you work exclusively with *real* input, you will get only *real* results with CPXRES clear (*startup default*); you can, however, also set CPXRES to allow for *complex* results. Try 1 +/- √x and see the different results.

With at least one *complex* input parameter in arithmetic operations or function calls, your *WP 43S* will set CPXRES automatically (indicated by C in the *status bar*).

With input closed for a *complex* ***x*** and POLAR clear, for example,…

- +/- will change the signs of both the *real* and the *imaginary* part (as shown above),

---

137 Choosing rectangular notation and multiplication dots allows for displaying *real* and *imaginary* components using large font within (10^{-999}, 10^{999}) in SCI 4 together. It will work in SCI 5 for both components within (10^{-99}, 10^{99}). Staying with the *startup default* (i.e. MULT×) instead will cost you one displayed decimal in *complex* domain.

In *polar display mode*, angles will be normalized to ($-\pi$, π] always, so there will be no space for a power of ten needed for an angle; hence this will allow for SCI 6 within (10^{-999}, 10^{999}) regardless of the multiplication symbol chosen, and for SCI 7 within (10^{-99}, 10^{99}). See the *ReM*.

- CPX conj will change the sign of the *imaginary* part only, and
- CPX Re⇄Im will swap *real* and *imaginary* parts.

Press ENTER↑ for separating *complex* input as you do in *real* domain.

Example (with startup default settings):

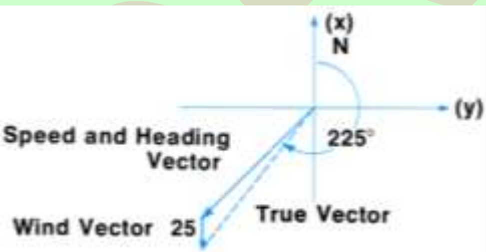
(**1** + **2** ***i***) × (**3** + **4** ***i***) is entered and solved like this:

1 CC **2** ENTER↑		1. +i×2.
3 CC **4**		3. +i×4_
×	returning	-5. +i×10.

Many transcendental functions will operate on *complex numbers* as well (e.g. sin, cos, tan, LN, e^x, y^x, $\sqrt{x}$, etc.). Please check pp. 161f.

Complex Numbers Used for 2D Vector Algebra

You can use *complex* domain for 2D vector algebra as demonstrated below. The functions |x|, +, –, CROSS, DOT, and UNITV wait for you – see the *menu* CPX and the *IOI*.



We can, for **example**, compute Mr. *Sweeney's* ground course (as explained on p. 130) according to the following alternative way:

DISP FIX 2

MODE DEG

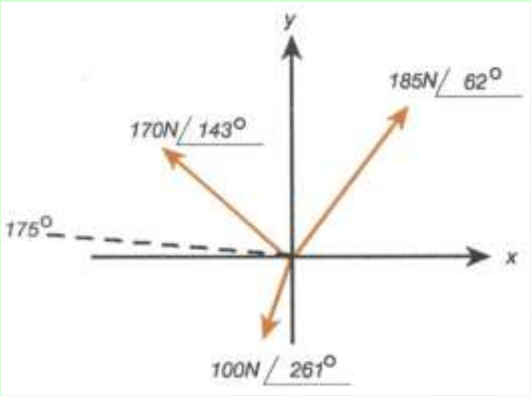
SF SYS.FL POLAR

125 CC **225**	125 ∡ 225_	indicated air speed and heading.
ENTER↑	125.00 ∡ 225.00°	
25 CC **180**	25 ∡ 180_	for the north wind.
+	143.77 ∡-142.06°	for the resulting vector

CC	r = 143.77 θ = -142.06°
360 +	143.77 217.94°

(according to navigational convention, the angle must be positive. – Compare with pp. 130f.)

Two **examples** more (taken from the *HP-42S OM*):



Dot Product of *Complex Numbers*

The figure … represents three two-dimensional force vectors. Use *complex numbers* and add the three vectors. Then use the DOT (dot product) function to find the component of the resulting vector along the 175° line.

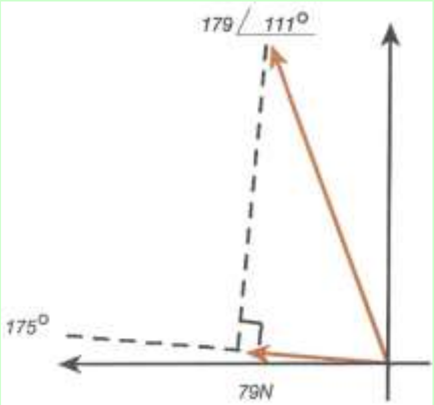
Solution:

MODE DEG	
SF SYS.FL POLAR	ensure proper *ADM* and coordinates.
170 CC 143 ENTER↑	125.00 ∡ 225.00°
185 CC 62	185 ∡ 62_
+	270.12 ∡ 100.43°
100 CC 261	100 ∡ 261_
+	178.94 ∡ 111.15°

Now, take the unit vector at 175°:

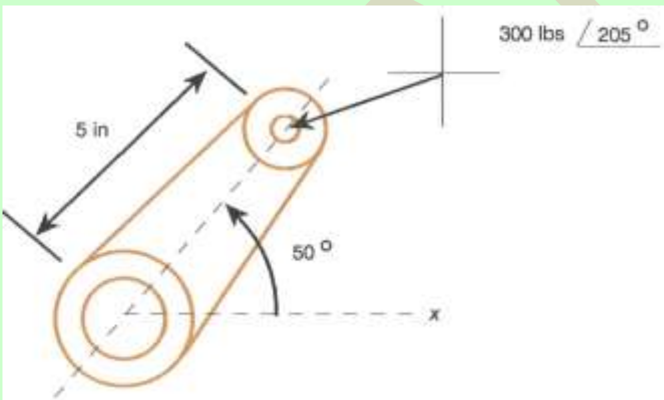
1 CC 175	1 ∡ 175_
CPX dot	78.86

Thus, the resulting vector sum has a component of approximately 79 *Newtons* in the direction of 175°. See the drawing overleaf.



Computing Moments.

To compute the moment of two vectors, use the CROSS (cross product) function. The cross product of two vectors is a third orthogonal vector. However, when two *complex numbers* are crossed, the *WP 43S* simply returns a *real number* that is equal to the signed magnitude of the resulting moment vector.



Find the moment generated by the force acting through the lever in the illustration below, where

$$\vec{M} = \vec{r} \times \vec{F}$$

Note this picture shows a two-dimensional situation. Lever and force are both acting in the drawing plane.

Solution:

MODE DEG	
SF SYS.FL POLAR	ensure proper *ADM* and coordinates.

Key in the radius vector and the force vector:

5 CC **50** ENTER↑	5.00 ∡ 50.00°
300 CC **205**	300.00 ∡ 205_
CPX cross	633.93

The moment vector has a magnitude of 634 *pounds* times *inches* and, since the result is positive, the vector points up, perpendicular to the plane of this page.[138]

Complex Numbers: Summary of Functions

Many of the numeric functions operating on *reals* also work for *complex numbers*:

- General mathematics:
  - *Monadic* functions:

    $\sqrt{x}$ and x^2, $\sqrt[3]{x}$ and x^3, 2^x and lb x, 10^x and lg, 1/x, e^x and ln, sinh, cosh, and tanh as well as their inverses work as usual; the same applies to sin, cos, tan, and their inverses (cf. also pp. 125ff for more information about angular I/O),

    e^x-1 and ln(1+x) return more accurate results with $x \approx 0$,

    +/- returns $x \times (-1)$ (a.k.a. 'unary minus') for closed input and POLAR clear while it turns x by 180° for POLAR set, and

    $(-1)^x$ returns $cos(\pi x)$ for non-integer x.

  - *Dyadic* functions:

    +, −, ×, /, y^x and $\sqrt[x]{y}$ work as usual,

    $\log_x y$ calculates the logarithm of y for base x,

    dot and cross allow using *complex numbers* for 2D vector computations, and

    || returns $\left(\frac{1}{x}+\frac{1}{y}\right)^{-1}$ for $x \times y \neq 0$ and 0 else.

[138] If the problem you're working requires a true (three-dimensional) vector as a result, use a 1 × 3 or 3 × 1 matrix to represent each vector in three dimensions. See next chapters.

- Isolating and manipulating parts of *complex numbers*:

  Use CC for composing and cutting,

  RE for isolating the *real* part of x and IM for its *imaginary* part,

  Re⇄Im for swapping its *real* and *imaginary* part,

  |x| for the *magnitude* of x and ∡ for its *phase* (a.k.a. *argument*),

  FP for the fractional part of x and IP for its integer part;

  sign and UNITV return the unit vector of x, and

  conj returns its *complex conjugate*.

- Rounding:

  RDP ***n*** rounds x to n decimal places in FIX format
  (e.g. **1.23456789E−95** RDP **99** will return 1.2346×10^{-95}),

  ROUND rounds x using the current display format (like RND did on *HP-42S*), and

  RSD ***n*** rounds x to n significant digits.

- Advanced mathematics (see the *ReM, App. H* for comprehensive information about the functions following):

  - *Monadic* functions:

    FIB returns the extended *Fibonacci number*,

    g_d and g_d^{-1} the *Gudermann function* and its inverse,

    sinc returns $\frac{sin(x)}{x}$ for $x \neq 0$ and 1 for $x = 0$ (input shall be supplied in *radians* – cf. pp. 125f),

    W_p returns the principal branch of *Lambert's* W for $x \geq -1/e$,

    W^{-1} returns x for given W_p (≥ −1),

    x! ($= \Gamma(x+1)$) and Γ(x) calculate the *complex Gamma function*, and

    lnΓ returns its natural logarithm.

- *Dyadic* functions:

  AGM returns the *arithmetic-geometric mean*,

  COMB and PERM calculate with *complex Gamma*,

  β(x,y) returns *Euler's Beta function*, and

  lnβ its natural logarithm.

Vectors and Matrices: Introduction and Input

So far, we dealt with just one or two or (seldom) three numbers at once. Your *WP 43S* can do more for you – e.g. manipulate a set of numbers in a column or a row or even in an array of 4, 6, 8, 9, 10, 12, or more numbers simultaneously. Such number columns or rows are called *vectors* and the arrays are called *matrices* by mathematicians. If you do not know of vectors and matrices yet, feel free to set them aside; your *WP 43S* will serve you perfectly without them.

If you know of them, however, note the function set of your *WP 43S* covers vector operations and also allows for adding, multiplying, inverting, and transposing matrices, as well as for editing and manipulating parts of such matrices. It also provides functions for computing *determinants*, *eigenvalues* and *eigenvectors*, and for solving *systems of linear equations*. Its function set is based on the one of *HP-42S* and extends it.

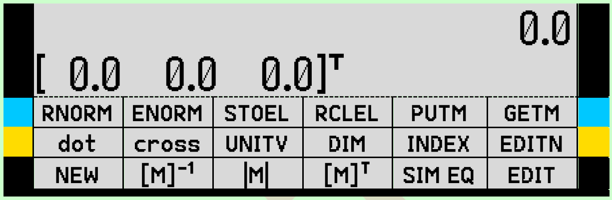
Generally, we talk of an ***n***×***m*** *matrix* if it features ***n*** rows and ***m*** columns. A vector may be regarded as a special matrix featuring one column or one row only.

Example:

A vector $\begin{bmatrix} 4 \\ -5 \\ 6.7 \end{bmatrix}$ and a matrix $\begin{bmatrix} -1 & 12 & 7 \\ 25 & 0 & 3 \end{bmatrix}$ shall be entered subsequently. The *stack* shall be clear at beginning.

Enter DISP FIX 0 1
3 ENTER↑ 1 MATX NEW (the leftmost unshifted *softkey*)

to initialize the 3D column vector (i.e. a 3×1 matrix). See the new matrix in **X** and the top *view* of MATX displayed in the *menu section*:



For saving screen space, your *WP 43S* displays each column vector *transposed* (thus the superscript T trailing it), i.e. in one row instead of one column on the screen. The vector is initialized with all its components being zero. To enter the vector components, press EDIT (the rightmost unshifted *softkey*) and the *Matrix Editor* will appear in the *menu section*:

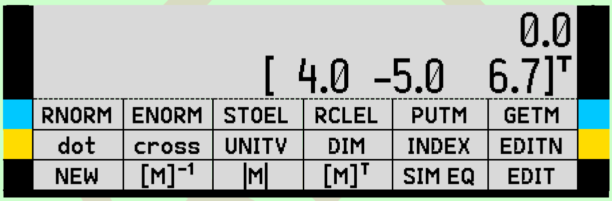
Note the 1st element of the vector is displayed inverted now indicating the position of the edit cursor. This particular element is shown below in the format set (i.e. FIX 1 here), so we need two rows for **X**.

Now press 4

Move the cursor to the next element: →

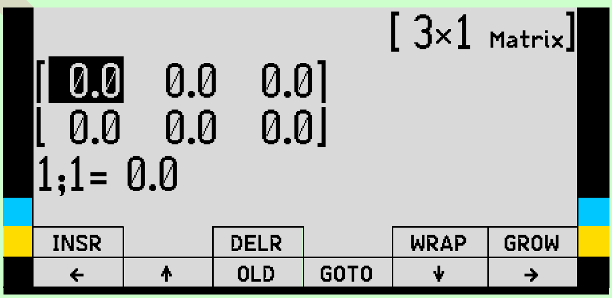
Continue editing: **5** +/- → **6.7**

EXIT



Note EXIT left the *Matrix Editor*, returning to the top *view* of MATX, closes input for the object in **X**, and shifts ***x*** to the right.

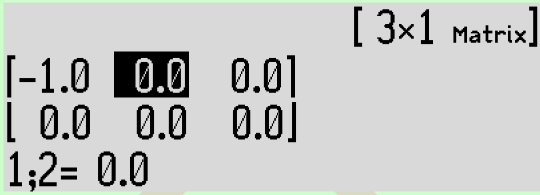
Now, let's initialize the 2×3 matrix via
2 ENTER↑ **3** NEW and begin editing once again by
EDIT



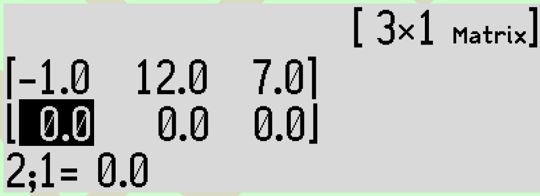
Three numeric rows are required for editing ***x*** now. The 3×1 matrix in **Y** above is the 3D vector we just entered before; note any matrix is displayed in this short form (with a × even for MULT· chosen) in any *stack register* but **X**.

Again, all elements of the new matrix start containing zero. Its 1st element is displayed inverted as the 1st element of the vector was above. Matrix editing will continue in analogy:

1 +/- →

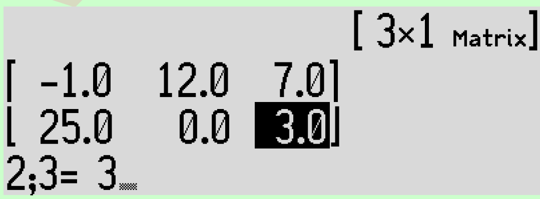


12 → **7** →

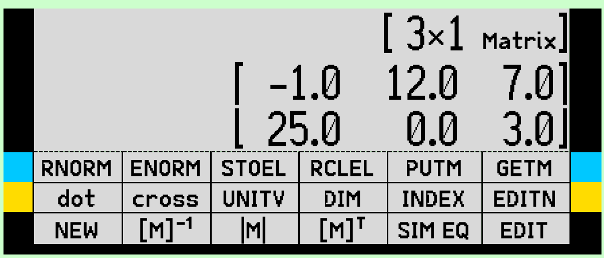


Entering the last → moved the cursor from the last element of row 1 to the 1st element of row 2. So you can simply continue row-wise:

25 → → **3**



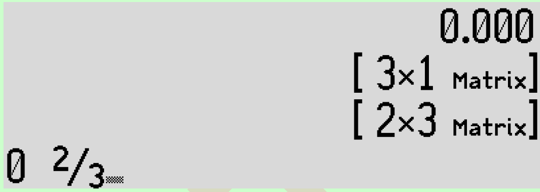
EXIT



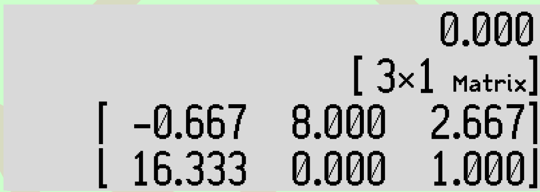
Now also this matrix is closed and ready for calculating. Assume you want to multiply it by $^2/_3$; and you want more than just one decimal displayed in the result:

DISP FIX 3

. 2 . 3



Press × and you will get immediately



which are all matrix elements multiplied by $^2/_3$ at once.

You may store such matrices in any *register* or variable. So let's store our resulting matrix in **R00** – just press STO 0 0 for this.

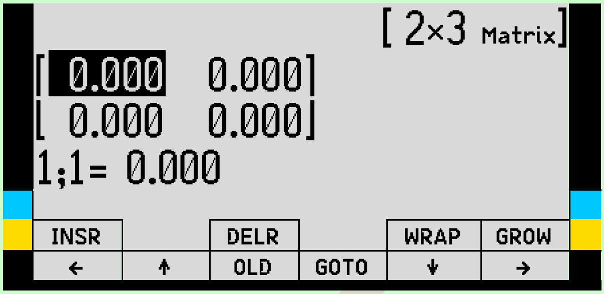
You can also create and fill a matrix directly in a variable (i.e. you do not have to create the matrix on the *stack* and store it afterwards).

Example:

Create a quadratic matrix $[MA] = \begin{bmatrix} 4 & -3 \\ -2 & 1 \end{bmatrix}$ and fill it directly.

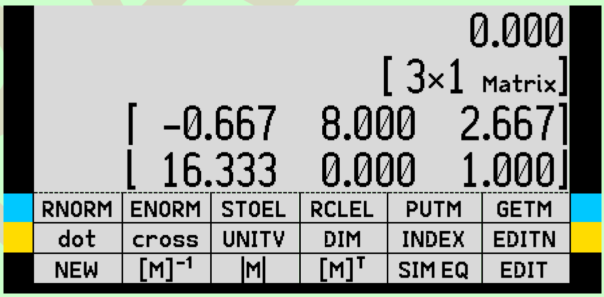
2 ENTER↑ DIM α M A ENTER↑ creates **MA** as a 2×2 matrix.

EDITN VAR MA



4 → **3** +/- → **2** +/- → **1**

Now, press EXIT and you are done with **MA** – while the screen looks just as before again:



Vectors and Matrices: Displaying and Editing Larger Objects

Whenever **X** contains a matrix, your *WP 43S* will try to show it completely (i.e. display all its elements in the format you chose for *reals*). Objects in higher *stack registers* will be indicated in a single row (abbreviated if necessary) or will be shifted out of the display window – but *x* will stay on the screen at least.

If space does not suffice for showing the complete matrix in the format chosen, your *WP 43S* will switch to the small font automatically.

Example (continued):

DISP FIX 5

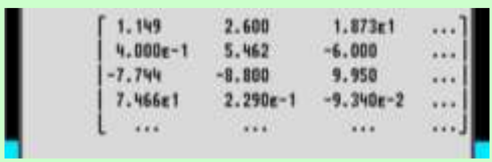
If font switching should not suffice, your *WP 43S* will furthermore automatically turn to abbreviated SCI **3** for the elements of the respective matrix. This allows for showing arbitrary 5×4 *real* matrices entirely. If a *real* matrix exceeds five rows, its fifth row is displayed filled with ellipses (**…**); if it exceeds four columns, its fourth column is shown filled with ellipses.

Example:

Assume a 6×5 matrix

$$x = \begin{bmatrix} 1.1493 & 2.6 & 18.725 & 3 & 9.2 \\ 0.4 & 5.462 & -6 & 95.1 & 51.6 \\ -7.744 & -8.8 & 9.95 & 54.5 & 0.17 \\ 74.66 & 0.229 & -0.0934 & 2 & -3.829 \\ 33.9 & -79.4 & 3.436 & 9.08 & 4.256 \\ 0.0488 & 7 & 5.98 & -0.68 & -22.492 \end{bmatrix}$$

was entered on the present *stack* and is in **X** now. Then the screen will look like this to scale:

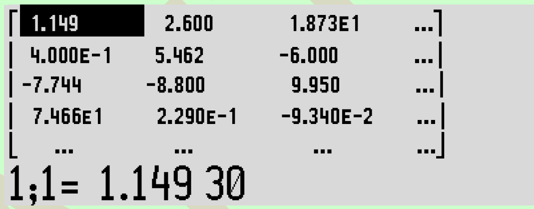


Editing such a large matrix will push also ***y*** from the screen until input is closed again. You can browse the entire matrix regardless of its size always.

For matrices larger than 5 rows and/or 4 columns, the display may vary depending on the cursor position: ellipses may appear on top and bottom, left and right side. A view of 3×3 matrix elements including the one selected by the cursor can be seen always at least – this selected element is also displayed below of the matrix in the format you have chosen for *reals*. Since the indices of this element are shown there as well you always know where you are.

Example (continued):

Press MATX EDIT and you will see:



The 1st matrix element is selected. And the lowest numeric row displays this element in FIX 5 as we had chosen.

Go to the bottom row of this matrix by pressing ▼ (or ↓) five times and you will get:

Now go to the very last element of this matrix by pressing → four times:

Wherever you are within a matrix, you can replace or modify the currently selected element in two ways:

1. Let an arbitrary *monadic* function operate on the selected element. If you need any *menus* to reach a function, they will temporarily replace the *Matrix Editor menu*; exiting those *menus* will bring you back to the *Matrix Editor menu*.
2. Simply key in a new number replacing the old one.

Example (continued):

Replace the last matrix element by 17.435.

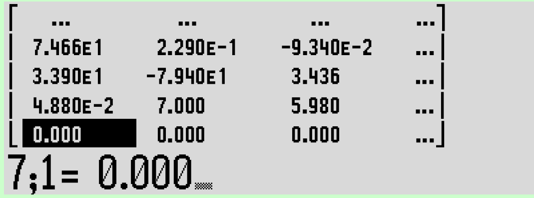
17.435

If you now decide you want to recover the old element again, however, call OLD. This old content is actually not overwritten until you press one of ▲, ↑, ▼, ↓, ←, or → after entering a new number, or you leave the *Matrix Editor* via EXIT.

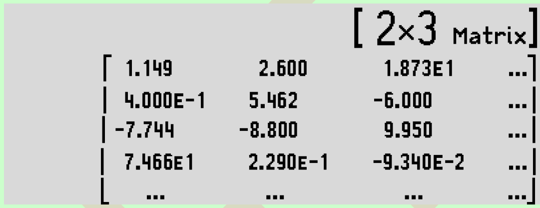
☞ Repeatedly pushing the cursor in one direction (e.g. by →) will jump from the 1st, 2nd, etc. to the last row and then return to the 1st row in default WRAP mode. If GROW is set instead, another → from the very last (i.e. bottom right) matrix element will add a new row to the matrix.

Example (continued):

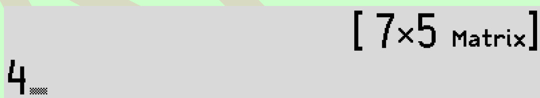
→



Here, we are done with that matrix for now. So press EXIT and you will see again:



Note the 1st matrix element is not highlighted anymore since you left the *Matrix Editor*. Thus, just entering 4 will display (due to *automatic stack lift*) now:



So matrix editing is easy and straightforward. The *IOI* contains additional information, also about the further commands DELR, INSR, and R⇆R showing up in the *Matrix Editor menu*.

Vectors and Matrices: Complex Stuff

Your *WP 43S* supports also *complex* vectors and matrices, i.e. matrices containing *complex* elements. They are created and initialized like *real* objects via NEW or DIM as explained above. Or you can recall a *real* matrix and edit it; if you enter one or more *complex numbers* for its elements it becomes a *complex* matrix – you can store it at the same or another place after editing.

Example (continuation of p. 168):

Create and store a *complex* matrix $\begin{bmatrix} 5+8i & \pi i \\ -2 & 4-3i \end{bmatrix}$.

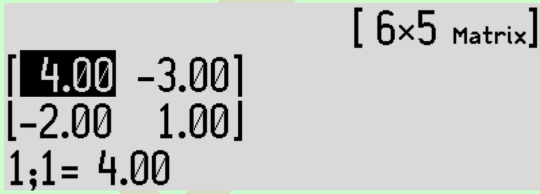
Solution:

Remember we have created a 2×2 matrix just a few pages ago. So it is most easy to recall it for using it as a template:

RCL VAR MA

DISP FIX 2 since this will suffice for the process following.

MATX EDIT



We can now just enter the new elements there as we have done before:

5 CC 8 → 0 CC π → → 4 CC 3 +/-

EXIT

STO 0 1

Since we edited on the stack and stored the resulting new *complex* matrix in a new location, the old *real* matrix **MA** is not affected at all.

Compare pp. 154f for the input and formatting of *complex numbers*. Everything else works as it does for *real* matrices. You see *complex* vectors and matrices are no complex topic at all for you with your *WP 43S*.

Vectors and Matrices: Calculating

As we have seen on p. 167, your *WP 43S* can multiply a matrix by a plain number (a.k.a. *scalar*); doing this, each element of said matrix is multiplied by said number. Additions, subtractions, and divisions work alike for a matrix y combined with a scalar x. Vice versa, with a scalar y and a matrix x, additions, subtractions and multiplications will work the same way (remember you cannot divide a number by a matrix). Also *monadic* functions operate on each matrix element in your *WP 43S*, if applicable.

Examples:

With an arbitrary matrix in **X**, pressing...

- $\sqrt{x}$ will extract the square root of each matrix element individually (if CPXRES is set, a *real* matrix x containing at least one negative element will become *complex* this way).
- x^2 will square each matrix element individually (use ENTER↑ × for squaring the matrix instead);
- |x| will calculate the absolute value of each matrix element (instead, use MATX ENORM for calculating the *Euclidean* norm of the matrix or take |M| for getting its determinant);
- +/- will change the sign of each matrix element.

You can also let the *dyadic* functions +, −, ×, or / operate on two matrices or vectors alone (i.e. *data types* 8 and 9), provided the rules of *linear algebra* are obeyed:

	y	x	Op.	Resulting x
+	[$m \times n$ Matrix]	[$m \times n$ Matrix]	$[y]+[x]$	[$m \times n$ Matrix]
−			$[y]-[x]$	
×	[$m \times n$ Matrix]	[$n \times p$ Matrix]	$[y]\cdot[x]$	[$m \times p$ Matrix]
/	[$m \times n$ Matrix]	[$n \times n$ Matrix]	$[y]\cdot[x]^{-1}$	[$m \times n$ Matrix]

The 1st row of this table reads as follows: For adding or subtracting two arbitrary matrices, both must be of identical size, and the result will be of the same size as well. The subsequent rows read in analogy. [139] If either matrix is *complex*, the result will be *complex* in most cases as well.

Example (continuation of p. 168):

Multiply the matrices in **R00** and **MA**. Output format shall be FIX 3.

Solution (we omit the *menu section* in the following pictures):

DISP FIX 3

RCL 0 0

```
                [2×2 ℂ matrix]
[ -0.667   8.000   2.667]
[ 16.333   0.000   1.000]
```

Note the '**2×2 ℂ matrix**' in **Y** is the *complex* matrix we entered in previous chapter – the *stack* handles matrices as it handles other objects. Now let's recall **MA**:

RCL VAR MA (or RCL α M A ENTER↑, if you have defined many variables already – cf. p. 57)

```
     [2×3 Matrix]
[ 4.000 -3.000]
[-2.000  1.000]
```

The 2x3 matrix in **Y** now is the one we have recalled from **R00** into **X** before recalling **MA**. We multiply ***y*** times ***x*** as usual by

---

139 Remember matrix multiplication behaves different than multiplication of numbers. Generally, for two arbitrary matrices A and B of matching sizes, $A \cdot B \neq B \cdot A$. Also note that only square matrices can be squared.

And matrix division is special: it is defined as multiplication of the numerator times the inverse of the denominator. Therefore, **X** must contain a nonsingular (i.e. invertible) matrix here – else you cannot divide by that matrix. Only square matrices can be inverted.

[×] resulting in

```
              [ 2×2 ℂ matrix]
[-79.000  48.000  19.000]
[ 27.000 -24.000 -11.000]
```

You see that arithmetic operations on matrices are almost as easy as on scalars using your *WP 43S*.

And your *WP 43S* features further matrix operations: |M| for computing determinants, $[M]^{-1}$ for inverting, $[M]^T$ for transposing, M.LU for computing the LU decomposition, and two norms (*Euclid's* ENORM and the row norm RNORM) – please look them up in the *IOI*.

Example:

We want to invert a 2×2 matrix $[M] = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$.

Solution:

Just enter the matrix as usual

2 [ENTER↑] [MATX] [DIM] [α][M] [ENTER↑] creates **M** as a 2×2 matrix.

[RCL] [VAR] [M]

[EDIT] etc.

```
      [ 3×2 Matrix]
[ 1.000   2.000]
[ 3.000   4.000]
```

[[M]⁻¹]

```
      [ 3×2 Matrix]
[-2.000   1.000]
[ 1.500  -0.500]
```

Thus, the inverted matrix reads $[M]^{-1} = \begin{bmatrix} -2 & 1 \\ 1.5 & -0{,}5 \end{bmatrix}$.

For two **vectors** of identical size, there are two special multiplications provided: DOT and CROSS. DOT will return the dot product, a scalar – exactly what the table above says for ***m*** = ***p*** = 1. CROSS works for two 2D or 3D vectors and will return their cross product.

Example from the *HP-27 OH*:

The force $\vec{F}$ on a particle with charge q which is moving with a velocity $\vec{v}$ through a magnetic field $\vec{B}$ is given by $\vec{F} = q\,\vec{v} \times \vec{B}$. Suppose a proton ($q = -e = 1.6 \cdot 10^{-19}\,coulomb$) is moving with velocity $\vec{v} = (0.4 \quad 2.8 \quad -1.2) \cdot 10^{7}\ {}^{m}/_{s}$. A uniform magnetic field surrounding the proton is of a strength $\vec{B} = (1.3 \quad -0.3 \quad 0.7)\ tesla$. Calculate the force on the proton.

This can be written as

$$\vec{F} = q\,\vec{v} \times \vec{B}$$
$$= 1.6 \cdot 10^{-19} \cdot 10^{7} \cdot (0.4 \quad 2.8 \quad -1.2) \times (1.3 \quad -0.3 \quad 0.7)$$

Solution:

Just remember that in cross products, vectors must be entered in proper sequence as written from left to right:

DISP FIX 2 — since this will suffice for that process.

3 ENTER↑ 1 MATX DIM α V ENTER↑ creates **v** as a 3×1 matrix.

RCL VAR V

EDIT etc.

```
[2×2 Matrix]
[ 0.40  2.80 -1.20]
```

STO α B ENTER↑ — creates **B** as a 3×1 matrix, too.

RCL VAR B

EDIT etc.

```
[3×1 Matrix]
[ 1.30 -0.30  0.70]
```

cross

[2×2 Matrix]
[1.60 -1.84 -3.76]

E 7 ×
1.6 E +/- 19 × resulting in

[2×2 Matrix]
[2.56×10^{-12} -2.94×10^{-12} -6.02×10^{-12}]

... *newtons*, of course.

The total 'length' or absolute value of this force is

ENORM

[2×2 Matrix]
5.14×10^{-11}

Compare with the weight of a proton:

CONST α M P ␣	recall the proton mass m_p.
G ␣ ×	recall earth acceleration $g_⊕$ and get weight.

5.14×10^{-11}
1.64×10^{-26}

So this is a force ratio of

/

3.14×10^{15}

Thus, physicists deliberately neglect gravitational effects in such microscopic calculations.

If you just want to perform elementary vector operations in 2D, however, there are two simple alternatives (known for long from earlier calculators):

1. Enter the Cartesian components of each vector in **X** and **Y** (if necessary, converting its polar components into Cartesian ones by

R← before) and use Σ+ for additions or Σ− for subtractions. Recall the result via SUM ; it may look like this, for example:

```
Σy =     1 464.21
Σx =       123.58
```

2. Calculate with *complex numbers* (cf. pp. 154ff). In *complex* domain, 2D vector multiplication is possible since the commands DOT and CROSS are contained in CPX as well. Cf. pp. 158ff for examples.

Vectors and Matrices: Solving Systems of Linear Equations

Your *WP 43S* can also solve simultaneous linear equations (of the kind $[A] \cdot \vec{X} = \vec{B}$) for you.[140] To deal with such a system of linear equations, proceed as follows:

1. Specify the number of unknowns (e.g. 4) by entering

   MATX SIM EQ 4

   Your *WP 43S* automatically creates (if necessary) and dimensions three matrices: MatA, MatB, and MatX. You will see a new *menu* showing up:

Mat A	Mat B				Mat X

2. Press **Mat A**. The *Matrix Editor* will open and you can enter the elements of the 4×4 *coefficient matrix* (see on pp. 163ff how to do this). Close the *Matrix Editor* by EXIT to return to the *menu* shown above.

3. Press **Mat B** and enter the elements of the 4×1 *constant matrix* the same way (this is a vector actually).

[140] This works the same way as it did on the *HP-42S*. The number of unknowns is only limited by the free memory available in your *WP 43S* at execution time.

4. Press Mat X to let your *WP 43S* compute the 4×1 *solution matrix* (a vector again). You are done!

To work another problem with the same number of unknowns, return to step 2 or 3. For a problem with a different number of unknowns, press EXIT and start over with step 1.

Vectors and Matrices: Eigenvalues and Eigenvectors

An *eigenvalue* is a *real* or *complex number* λ solving the matrix equation $[A] \cdot \vec{X} = \lambda \cdot \vec{X}$. Then, the vector $\vec{X}$ is called an *eigenvector* of $[A]$.

Usually, there will be more than one λ and a multitude of vectors $\vec{X}$ solving this problem. Thus, the simplest set of linearly independent vectors $\vec{X}$ is chosen to build the base of the *eigenspace* belonging to a particular eigenvalue found. And the simplest set of eigenvectors building a base of a space of the same dimension as $\vec{X}$ are called <u>the</u> eigenvectors of $[A]$.

Your *WP 43S* can solve such problems for you as well:

Example 1:

We need the eigenvalues of a matrix $[M] = \begin{bmatrix} 2 & 1 \\ 6 & 1 \end{bmatrix}$.

Solution:

We have got a 2×2 matrix named **M** already. We don't need its old contents anymore so we simply recall and edit it:

RCL VAR M

DISP FIX 0 1

MATX EDIT etc.

```
[ 2.0   1.0]
[ 6.0   1.0]
```

STO VAR M

The eigenvalues are the solutions of the *characteristic polynomial* of this problem:

$$(2-\lambda)(1-\lambda)-6=0$$

▲ **EIGVAL** returns

```
M = [ 2×2 Matrix]
    [ 4.0   0.0]
    [ 0.0  -1.0]
```

being the matrix with the eigenvalues as its diagonal elements. Note this resulting diagonal matrix is pushed on the *stack*.

Example (continued):

Now, what are the eigenvalues of $[N]=\begin{bmatrix}3 & 4\\ -4 & 3\end{bmatrix}$?

Solution:

▲ **EDIT** etc.

```
M = [ 2×2 Matrix]
    [ 3.0   4.0]
    [-4.0   3.0]
```

STO **α** **N** **ENTER↑**

▲ **EIGVAL** returns

```
M = [ 2×2 Matrix]
N = [ 2×2 Matrix]
[ 3.0 +i× 4.0   0.0         ]
[-4.0           3.0 +i× 4.0]
```

Note that although **N** contained only *real* elements, we get *complex* eigenvalues here.

Example 2:

What are the eigenvalues of $[Q]=\begin{bmatrix}0 & 0 & -2\\ 1 & 2 & 1\\ 1 & 0 & 3\end{bmatrix}$?

Solution:

3 ENTER↑ MATX DIM α Q ENTER↑ creates **Q** as a 3×3 matrix.

RCL VAR Q

EDIT etc.

```
             [ 2×2 ℂ Matrix ]
[ 0.0   0.0 -2.0]
| 1.0   2.0  1.0|
[ 1.0   0.0  3.0]
```

▲ EIGVAL returns

```
       Q = [ 3×3 Matrix]
[ 2.0   0.0  0.0]
| 0.0   2.0  0.0|
[ 0.0   0.0  1.0]
```

Note one eigenvalue comes twice here. Let's get the eigenvectors of **Q** now – they will be put out as a matrix whose rows are these vectors:

x≷y returns **Q** into **X**:

```
           [ 3×3 Matrix]
[ 0.0   0.0 -2.0]
| 1.0   2.0  1.0|
[ 1.0   0.0  3.0]
```

EIGVEC pushes this matrix on the *stack*:

```
       Q = [ 3×3 Matrix]
[ 1.0   0.0 -2.0]
| 0.0   1.0  1.0|
[-1.0   0.0  1.0]
```

STO α V ENTER↑

× returns

```
               [ 3×3 Matrix]
        [ 2.0   0.0  -2.0]
        | 0.0   2.0   1.0|
        [-2.0   0.0   1.0]
```

[RCL] [L] recalls **V**.

[▲] [M]$^{-1}$

[×] returns

```
               [ 3×3 Matrix]
        [ 2.0   0.0   0.0]
        | 0.0   2.0   0.0|
        [ 0.0   0.0   1.0]
```

This looks very much like what was returned for the eigenvalues of **Q** above. Let's check:

[DISP] FIX [4]

[−] returns

```
                                [ 2×2 ℂ Matrix]
[ 0.000 0   0.000 0   0.000 0]
| 0.000 0   0.000 0   0.000 0|
[ 0.000 0   0.000 0   0.000 0]
```

So the result of $[V]^{-1} \cdot [Q] \cdot [V]$ with **V** being the matrix of the eigenvectors of **Q** is exactly the diagonal matrix of the eigenvalues of **Q**.

Your *WP 43S* can compute eigenvalues and eigenvectors for matrices featuring rational elements as well:

Example 3:

What are the eigenvalues of $\begin{bmatrix} -38 & 43/7 & 63/2 & 1149/14 \\ -14 & 19/7 & 7 & 181/7 \\ -8/7 & -122/49 & 24/7 & 177/49 \\ -16 & 26/7 & 13 & 244/7 \end{bmatrix}$?

Solution:

4 ENTER↑ MATX NEW creates a 4×4 matrix.

EDIT 38 +/- → .43.7 → .63.2 → .1149.14 → etc.

Note each matrix element can be entered as integer or fraction but is converted to a *real number* following the current display settings as soon as said element is closed:

[3×3 Matrix]

-38.000 0	6.142 9	31.500 0	82.071 4
-14.000 0	2.714 3	7.000 0	25.857 1
-1.142 9	-2.489 8	3.428 6	3.612 2
-16.000 0	3.714 3	13.000 0	34.857 1

DISP FIX 0 1 shall suffice here:

-38.0	6.1	31.5	82.1
-14.0	2.7	7.0	25.9
-1.1	-2.5	3.4	3.6
-16.0	3.7	13.0	34.9

MATX ▲ EIGVAL returns:

-5.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	3.0	0.0
0.0	0.0	0.0	5.0

Note the 2nd eigenvalue is zero here.

RCL L

EIGVEC displays:

4.0	5.0	4.0	5.0
3.0	-2.0	2.0	-4.0
1.0	-4.0	-3.0	5.0
1.0	4.0	3.0	1.0

Generally, your *WP 43S* solves characteristic polynomials numerically.

Vectors and Matrices: Dealing with Statistical Data

We mentioned above you can enter 2D statistical data using a matrix as well as keying them in point after point. How is this done?

Let's return to the **application** introduced on p. 113 with its step 4 – remember there were 30 samples measured twice in a special way using the instrument under investigation, resulting in 30 pairs of measured values:

4. Create a 1×2 named matrix and open it for editing:

   1 ENTER↑ **2** DIM α M S A ENTER↑ creates **MSA** accordingly.

   EDITN VAR MSA

   GROW allows the matrix to grow with data entered.

   Now key in all 30 pairs of measured values. The 1st value shall be x, the 2nd be y – thus, the keystroke sequence will be ***mv1*** → ***mv2*** for each sample. A subsequent → lets the matrix grow by one row for the next data point.

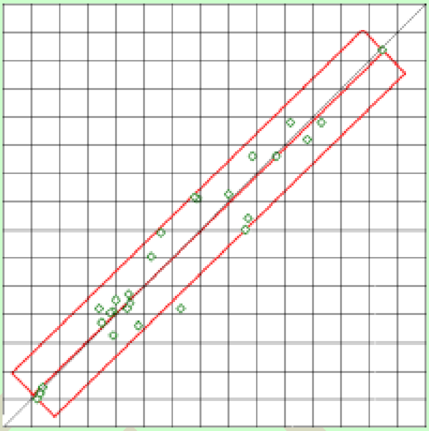
With all points entered, eventually key in

STAT CLΣ

RCL VAR MSA

Σ+ Calling Σ+ with a 30×2 matrix in **X** will display the data of the 30th data pair in **X** and **Y** (and save a copy of the data point matrix in **L**).

5. It is recommended to plot these 30 data points[141] (see a typical diagram here but check p. 114 and its footnote as well).



6. Let your *WP 43S* fit a straight line through the points and compute $c_0 = \frac{T}{30 s_x s_y}\sqrt{\frac{s_x^2 + r^2 s_y^2}{1-r^2}}$ with ***T*** being the width of the tolerance zone you want to control:

▲ OrthoF	select the orthogonal linear fit model.
r x^2 STO K	get the *correlation coefficient* and store its square.
▼ s x^2 R↓	get s_x^2 and roll it out of the way.
x^2 ×	get $r^2\, s_y^2$.
R↑	return s_x^2 from the top *stack register* and
+	calculate the numerator
1 RCL − K	and the denominator.
/ $\sqrt{x}$	this is the 2nd factor now.
s × / 30 /	divide by **30** $s_x\, s_y$.
.01 ×	this returns c_0 for our ***T*** now (cf. pp. 113ff).

If you get $c_0 \geq 1$ then this measuring device may be used for controlling the given tolerance zone under these conditions – else look for a more precise instrument, better measuring conditions, or a wider tolerance.

[141] Steps 5 and 6 are actually the same as shown in the application above with input of separate real numbers (instead of one matrix) already. They are just repeated here for sake of completeness.

Vectors and Matrices: Summary of Functions

Assume **X** contains a matrix. Then there are functions operating on ***x*** as a whole and others just operating on its elements individually. Let us list the first set first:

- General mathematics:
  - *Monadic* functions operating on the entire matrix ***x***:

    ENORM computes the *Euclidean* norm of ***x*** (i.e. a *real number*),

    RNORM computes the row norm of ***x*** (i.e. a *real number*),

    RSUM computes the row sum of ***x*** (i.e. a vector),

    |M| computes the determinant of ***x*** (i.e. a *real* or *complex number*),

    [M]ᵀ returns the transpose matrix of ***x***,

    [M]⁻¹ returns the inverse matrix of ***x***,

    EIGVAL returns the eigenvalues of ***x***, and EIGVEC its eigenvectors (cf. pp. 180ff), while

    UNITV returns the unit vector of ***x*** (see the ReM).

  - *Monadic* functions operating on each element x_{ij} of ***x*** individually:

    +/-, 1/x, |x|, ∡, √x and x², ∛x and x³, 2ˣ and lb x, eˣ and ln, 10ˣ and lg, sin, cos, tan, sinh, cosh, and tanh as well as their inverses work as explained for *real* and *complex numbers* above,

    eˣ-1 and ln 1+x return more accurate results with $x_{ij} \approx 0$;

    sinc returns a matrix containing $\frac{sin(x_{ij})}{x_{ij}}$ for $x_{ij} \neq 0$ and 1 for $x_{ij} = 0$ (input shall be supplied in *radians* – cf. pp. 125f),

    (-1)ˣ returns $cos(\pi\, x_{ij})$ for non-integer x_{ij}.

    RDP ***n*** rounds x_{ij} to ***n*** decimal places in FIX format, ROUND rounds x_{ij} using the current display format, and

RSD n rounds x_{ij} to n significant digits.

For *complex* matrices, conj returns a matrix with the *complex conjugates* of x_{ij}.

For *real* matrices,

ceil returns a matrix with the smallest integers ≥ x_{ij} and floor with the greatest integers ≤ x_{ij} ,

FP returns a matrix with the fractional parts of x_{ij} and IP with their integer parts, while

sign returns a matrix with each x_{ij} replaced by $signum(x_{ij})$.

- *Dyadic* functions operating on x and y:

  [+], [−], [×], and [/] work as explained on pp. 174ff,

  cross operates on two *real* 2D or 3D vectors of identical size as shown on pp. 177f, and

  dot operates on two matrices of identical size.

- *Dyadic* function operating on each element y_{ij} of y individually:

  $\log_x y$ calculates the logarithms of y_{ij} for base number x.

- Isolating and manipulating bulk parts of a *complex* matrix x: Use …

  CX→RE for cutting x in its two parts,

  RE→CX for composing x from its two parts,

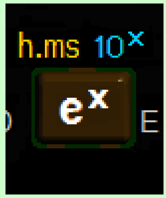
  Re for isolating its *real* part and Im for its *imaginary* part, and

  Re⇄Im for swapping its *real* and *imaginary* part.

The functionality of the Matrix Editor was demonstrated in the chapters above. Turn to the *ReM* for additional information about all matrix operations provided on your *WP 43S*. If you look for more general information about vectors and matrices, and further applications, please turn to a textbook covering *linear algebra*.

Times

There also is a special *data type* for time calculations on your *WP 43S*. Sexagesimal *times* are **entered** most easily in the format `hhhhh.mmssfff` terminated by h.ms – with `hhhhh` standing for *hours*, `mm` for *minutes*, `ss` for *seconds*, and `fff` for decimal fractions of *seconds* (these fractions may take more or less than three digits).



Example (with *startup default* settings):

Enter 5 *hours*, 39 *minutes*, and 7.8642 *seconds*:

5 . **39078642** h.ms

This is displayed with *startup default* settings: 5:39:07.864 2

Choosing CLK ▲ TDISP 2 will return 5:39:07

instead, while CLK ▲ TDISP 0 returns 5:39

The latter two formats allow for compact time displays like seen in digital clocks or watches. Note there is no rounding of *hours*, *minutes*, or *seconds* for *times*.

The colon is the unambiguous indicator for a *time* on your *WP 43S*. In general, there may be leading zeroes in the *minutes* and *seconds* sections and a settable number of digits after the 2nd colon. You can choose 12- or 24-hour display for *time of day*.

Example (continued):

Call TIME in the evening and you might get 21:47

FLAGS CF SYS.FL TDM24 9:47p.m.

☞ When *time of day* is returned by a function, it will be displayed according to your choice – internally, however, it is stored as standard 24-hour *time* for further calculations.

You may add and subtract sexagesimal *time intervals* simply using + and −; and *time intervals* may be multiplied or divided by any integer, rational, or *real number* – the result will stay a *time*. If you add an integer, rational or *real number* to such a *time*, it will be automatically converted to a *time* before adding. This applies to subtractions in analogy. Compare the matrices on pp. 71f.

Example (with *startup default* settings):

To meet your date at 5:25 p.m. at Stanford, you need 15' from your office to get your car out of the parking garage, 1.5 *hours* for the ride, and 12' for walking from the parking lot to lecture hall. Being careful, you count in another quarter of an hour for a possible traffic jam on the expressway. When do you have to leave your office?

Solution:

CLK ▲ TDISP 0

.15 h.ms **1.5**	+	returns	1:45
.12 h.ms	+		1:57
.1.4	+		2:12
5.25 h.ms	x⋛y −		3:13. So you have to leave at 3:13 p.m. the latest.

Note your *WP 43S* returns something looking like a 12h-time here even with *startup default* settings because it cannot know better based on the input given.

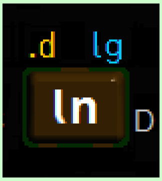
You can convert such (closed) sexagesimal *times* to decimal numbers using .d, and reconvert the decimal result to sexagesimal *times* by pressing h.ms.

There is only one more dedicated *time* command – SETTIM, serving obvious purposes.[142] GAP, ALL, ENG, FIX, or SCI have no effect on *times*.

---

[142] Note the real time clock in your *WP 43S* may deviate from true time by up to one minute per month (i.e. ± 25 ppm approximately, caused by parts tolerances; you live with this wearing a quartz watch as well – mechanical watches are less accurate generally). This deviation does neither affect real-world time calculations nor the TIMER application described in *Section 5*. If you are accustomed to radio controlled timepieces, however, you might find regular adjustments necessary.

Dates



For date calculations, choose one out of three *date* ***display*** *modes* (*DDM*) on your *WP 43S*: Y.MD, D.MY, and M.DY (these mode-setting commands are contained in CLK). *ISO* Y.MD is *startup default*.

Date **input** is decimal according to the *DDM* chosen and is terminated by .d (as shown on pp. 68f).

Example:

The 18th of December in 2017 is entered

2017.1218	.d	in Y.MD,
18.122017	.d	in D.MY, and
12.182017	.d	in M.DY.

Alternatively, any *real number* may be converted into a *date* via CLK x→DATE, and any triple of *reals* or integers via →DATE (cf. p. 40). Input containing more than the necessary digits for a *date* in the *DDM* selected will be rounded.

Vice versa, DATE→ splits a *date* in three integers and pushes them on the stack as demonstrated on p. 40. Note that both DATE→ and →DATE observe the *DDM* chosen. If you want to extract particular information from a *date* independent of current *DDM*, we recommend using one of the operations DAY, MONTH, or YEAR.

Like in the *status bar*, a closed *date* input or *dates* returned by a function are displayed as in the following **example:**

2017-12-18	in Y.MD,
18.12.2017	in D.MY,
12/18/2017	in M.DY.

So you immediately know the effective *DDM* from looking at the date format in the *status bar*.

CLK WDAY takes a *date* from the *stack* – or a decimal input of e.g. 2013.0504 in Y.MD mode (equivalent to inputs of 4.052013 in D.MY or 5.042013 in M.DY) – and returns an integer indicating the position of this day in the corresponding week, temporarily headed by the name of this weekday:

Saturday 6 .[143]

Expect similar returns after CLK DATE .[144]

There is only one more dedicated *date* command – SETDAT, serving obvious purposes.

Note integers (or the integer parts of *real numbers*) may be added to or subtracted from a *date*, always representing an integer number of days regardless of the *date* format set. And *dates* may be subtracted from *dates*, resulting in an integer number of days.

But that's it – these are all the legal operations. Compare the matrices on pp. 71f. GAP, ALL, ENG, FIX, or SCI have no effect on *dates*.

---

[143] Translator's note: Numeric output of WDAY corresponds to Chinese weekdays 1 to 6 directly. For Portuguese weekdays ('segunda-feira' etc.), add 1 to days 1 to 5.

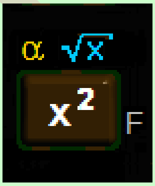
[144] Calculation of weekdays for the past depends on the calendars used at that time – there may be different true results for different countries depending on the date the particular country introduced the *Gregorian calendar*. Officially, that calendar became effective in 1582-10-15 in the catholic world. Large parts of the world took their time and switched later (see the chapter *Localizing Calculator Output* above and check *Wikipedia* for the dates applicable). Note, however, there are still also other calendars in use on this planet today, e.g. in the Muslim world.

Dates before the year 8 A.D. may be indicated differently than they were experienced at the time due to the inconsistent application of the leap year rule before. We count on your understanding and hope this shortcoming will not affect too many calculations.

Note that 8 A.D. should be written A.D. 8 or even better A.D. VIII instead – quite some false Latin is found in the English language. Nobody, however, counted years this way at that time – around the Mediterranean Sea, it was the year DCCLXI A.V.C. in best case (actually, this notation was broadly introduced some XL – or even CD – years later). Also note the *Julian calendar* was introduced and became valid not earlier than DCCVIII A.V.C. – before, months were organized differently. *Julius Caesar* was daggered in DCCIX A.V.C.; calendars may be a sensitive topic.

Alpha Input Mode : Introduction and Virtual Keyboard

This mode is designed for text entry, e.g. for keying in messages, prompts, and answers. It is entered via α typically. Within *AIM*, …



1. primary function of most keys will be appending the letter printed bottom right of the respective key to x – see the *virtual keyboard* overleaf;
2. the *menu* Myα will pop up immediately in the *menu section* (unless another menu is open), containing your favorite special characters or groups of them (up to 18 *items*);[145]
3. prefix g leads to homophonic Greek letters.[146]

Upper and lower case are set by ▲ and ▼, respectively, applying also to the letters in Myα and CATALOG'CHARS'αINTL (see pp. 195f).

Wherever a default primary function is not primary anymore in *AIM* but continues being meaningful, it is reached via *prefix* f. Thus, f is required here for appending a digit to x, for example. And f is also a shortcut to some special characters, like f +/_ calling ±.

---

[145] For people writing German, for example, Myα might look like pictured overleaf. Feel free to put other letters in - see *Section 6* for learning how to populate Myα.

[146] This will work wherever applicable, with "homophonic" following classical Greek pronunciation. Kudos to *Thales*, *Pythagoras*, *Heraclitus*, *Leucippus*, *Democritus*, *Aristotle*, *Archimedes*, *Euclid* (i.e. ὁ Θαλῆς ὁ Μιλήσιος, ὁ Πυθαγόρας ὁ Σάμιος, ὁ Ἡράκλειτος ὁ Ἐφέσιος, ὁ Λεύκιππος, ὁ Δημόκριτος, ὁ Ἀριστοτέλης, ὁ Ἀρχιμήδης ὁ Συρακούσιος καὶ ὁ Εὐκλείδης), and their colleagues for laying the foundations of logics, mathematics, and physics (i.e. τῆς λογικῆς και μαθηματικῆς τέχνης και τῆς φυσικῆς ἐπιστήμης) as we know them today – starting some 2600 years ago (note that the first two were called "practical arts" and the latter "theoretical science"). And kudos to the unnamed Babylonian mathematicians who built the foundations for these Greeks, actually recording e.g. what we now call "*Pythagoras' theorem*" 1200 years before *Pythagoras*.

We assigned **Gamma** also to C following the alphabet, and **Chi** to H since this Latin letter comes next in pronunciation. Three Greek letters require special handling since they lack single-letter equivalents in English: **Psi** is accessed via g 2 (since *looking* like **w** in a way), **Theta** via g U (following **T** corresponding to **Tau**), and **Eta** via g J. These three letters are printed in blue on the keyboard as reminders. **Omicron** is not featured since looking exactly like the Latin letter **O** in either case.

There is an '*alpha helper*' printed on the calculator back supporting users challenged by Greek.

Two extra prefixes operate exclusively in *AIM*: f R↓ makes the next directly keyboard-accessible input character a subscript if provided, while f E makes it a superscript. See the yellow arrows printed to the right of these two keys, above I and M.

And three alpha *menus* become accessible for more letters, punctuation marks as well as mathematical and other symbols (abbreviations are printed in blue and gold on the keyboard as reminders). Look up their contents in *Section 2* of the *ReM*, and use FBR to browse the entire character set provided.

Alpha Input Mode: Entering Simple Text and More

Your *WP 43S* features a large font for mainly numeric output and a small alphanumeric font for text strings. See here all characters directly evocable through the *virtual alpha keyboard* shown above:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

Α Β Γ Δ Ε Ζ Η Θ Ι Κ Λ Μ Ν Ξ Ο Π Ρ Σ Τ Υ Φ Χ Ψ Ω

α β γ δ ε ζ η ϑ ι κ λ μ ν ξ ο π ρ σ τ υ φ χ ψ ω

0 1 2 3 4 5 6 7 8 9   + −   × or · / . , ? ≷ ± # 🖶

and subscripts A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e i j k l m n o p q s u w v w x y z   α δ μ

and 0 1 2 3 4 5 6 7 8 9 + −

as well as superscripts a f g h o r T x and 0 1 2 3 4 5 6 7 8 9 + −

The 26 plain Latin letters can be also found in CATALOG'CHARS'αINTL (together with 73 more supporting international communication), the 24 basic (plus eleven accented) Greek letters in CATALOG'CHARS'A...Ω. αINTL can be called via **f** **A** in *AIM*; see the *ReM* for its content.

> So you may, for **example**, easily store and display an actual modern Greek message like
>
> Οι μελλοθάνατοι σε χάιρετουν.

Actually, we could have written the major part of this manual just using said small font. It covers at least 47 languages from Afrikaans to Zhōngwén (see the *ReM*), providing the means that your display messages or prompts can be easily read and understood by more than 50% of all mankind.

Example:

You can even store Dèng Xiǎopíng's famous and successful slogan

> Bùguǎn bái māo, hēi māo, dàizhù lǎoshǔ jiù shì hǎo māo!

in Pinyin straight ahead.

Taking advantage of this character set, it is also absolutely easy spelling e.g. French, Spanish, or German prompts correctly «en français», "en Español", or „auf Deutsch", as well as text strings in many more languages using letter sets based on Latin alphabets.

Your WP 43S supports you in climbing the very first step of politeness and respect by allowing you to adapt the software you write to the language your customers speak - instead of hacking in everything in English or using merely the very meager plain Latin letter set.

Two more *menus* (αMATH, called via g −, and α•, called via g .) contain further mathematical and non-mathematical symbols and marks (see the *ReM*):

! ¡ ¿ " ' § $ £ ¥ € % & ( ) * < = > @ [ ] \ ^ _ { } | ~

℅ « » ¬ ± · ÷ ← ↑ → ↓ ↕ ø ∞ ∟ ∡ ‖ ∧ ∨ ∫ ≈ := ≘ ≙ ≠ ≤ ≥

⊙ ⊥ 🕓 ℝ ℂ ⁻¹ ᵀ * E ∞ ⊙ ⊕ etc.

Pressing USER in *AIM* toggles USERα. Individual characters may be assigned to particular locations on the keyboard or within *menus* in *AIM* only (see pp. 290ff for how to do that). Such user assignments will become accessible when USERα is set (indicated by U and A or α being both lit in the *status bar*).

Alpha input can be edited character by character (like numeric input can be edited digit by digit) using ⬅ as long as it is open still.

AIM is closed by ENTER↑ (duplicating string ***x*** in ***y***) or by EXIT unless pressed in a *menu*. Empty strings will not be pushed on the *stack*.

Example (continued):

Pressing EXIT with Dèng Xiǎopíng's slogan keyed in will display it in **X** as shown above.

Pressing ENTER↑ instead will display the text twice – fully in **X** and shortened to one line in **Y**, showing only its first seven words trailed by an ellipsis.

Alpha strings exceeding two lines will show all their contents after SHOW only.

Combining Alpha Strings and Numeric Data

Due to the *data type* concept of your *WP 43S*, adding numeric data to a text string is as simple as pressing +.

Example:

Assume the two lowest *stack registers* look like this:

```
The train will arrive at
                   23:55
```

So here is an *alpha string* in **Y** and a *time* in **X**. Pressing + now will combine ***x*** and ***y***, returning

```
The train will arrive at 23:55
```

So, ***x*** will be converted to a string, taking into account its present display format, and will be appended to ***y*** (cf. the matrix on p. 71). Let's enter a 2nd string now by pressing α and the necessary characters, starting with a blank:

```
          The train will arrive at 23:55
sharp at Victoria station.
```

Leave *AIM* and close ***x*** by pressing EXIT:

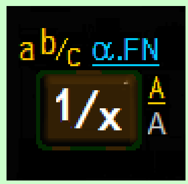
```
          The train will arrive at 23:55
              sharp at Victoria station.
```

Now we have got *alpha strings* in **X** as well as in **Y**, so pressing + once more will append ***x*** to ***y*** returning

```
    The train will arrive at 23:55 sharp at
                        Victoria station.
```

Strings may contain up to 196 characters in total. Once numeric data (like a *time* here) became part of an *alpha string*, they are fixed and will not vary even if format is changed. Easy, isn't it?

Working with Alphanumeric Strings



Your *WP 43S* provides some commands more for dealing with such strings. You find them all in α.FN:

αLENG? ***source*** pushes the length of the string in the source on the *stack*.

αPOS? ***source*** searches the string in the source for the character or string given in **X**; if a match is found, αPOS? returns the position number where the target was found starting (counting the leftmost character as position 0) – else it returns −1. Previous ***x*** is saved in **L**.

αRL ***source*** rotates the source string by ***x*** characters to the left.

αRR ***source*** rotates the source string by ***x*** characters to the right.

α→x ***source*** converts the leftmost character in the source to the corresponding code, removes this character from the source string, and pushes its code on the stack; if the source is empty, α→x returns zero.

x→α ***destination*** converts a character code in **X** to the corresponding character and appends it to the destination; the character code is saved in **L**.

If **X** contains an *alphanumeric string*, the entire string is appended to the destination.

If **X** contains a matrix, x→α uses each element in the matrix as a character code or *alphanumeric string*. x→α begins with the 1st element (1; 1) and continues row-wise (to the right) until reaching the end of the matrix.

αSL ***source*** shifts the source string by ***x*** characters to the left, deleting the first ***x*** characters from the string.

αSR ***source*** shifts the source string by ***x*** characters to the right, deleting the last ***x*** characters from the string.

You can also compare strings using commands in TEST to create something like a sorted list. A string (A) is called "smaller" than another one (B) if it precedes (B) in sorting.

Nevertheless, do not forget that your *WP 43S* is mainly designed as a programmable calculator. Please turn overleaf to see what can be performed with such a device.

HP-65 in space with Apollo-Soyuz.

The American astronauts calculated critical course-correction maneuvers on their HP-65 programmable hand-held during the rendezvous of the U.S. and Russian spacecraft.

Twenty-four minutes before the rendezvous in space, when the Apollo and Soyuz were 12 miles apart, the American astronauts corrected their course to place their spacecraft into the same orbit as the Russian craft. Twelve minutes later, they made a second positioning maneuver just prior to braking, and coasted in to linkup.

In both cases, the Apollo astronauts made the course-correction calculations on their HP-65. Had the on-board computer failed, the spacecraft not being in communication with ground stations at the time, the HP-65 would have been the only way to make all the critical calculations. Using complex programs of nearly 1000 steps written by NASA scientists and pre-recorded on magnetic program cards, the astronauts made the calculations automatically, quickly, and with ten-digit accuracy.

The HP-65 also served as a backup for Apollo's on-board computer for two earlier maneuvers. Its answers provided a confidence-boosting double-check on the coelliptic (85 mile) maneuver, and the terminal phase initiation (22 mile) maneuver, which placed Apollo on an intercept trajectory with the Russian craft.

Periodically throughout their joint mission, the Apollo astronauts also used the HP-65 to calculate how to point a high-gain antenna precisely at an orbiting satellite to assure the best possible ground communications.

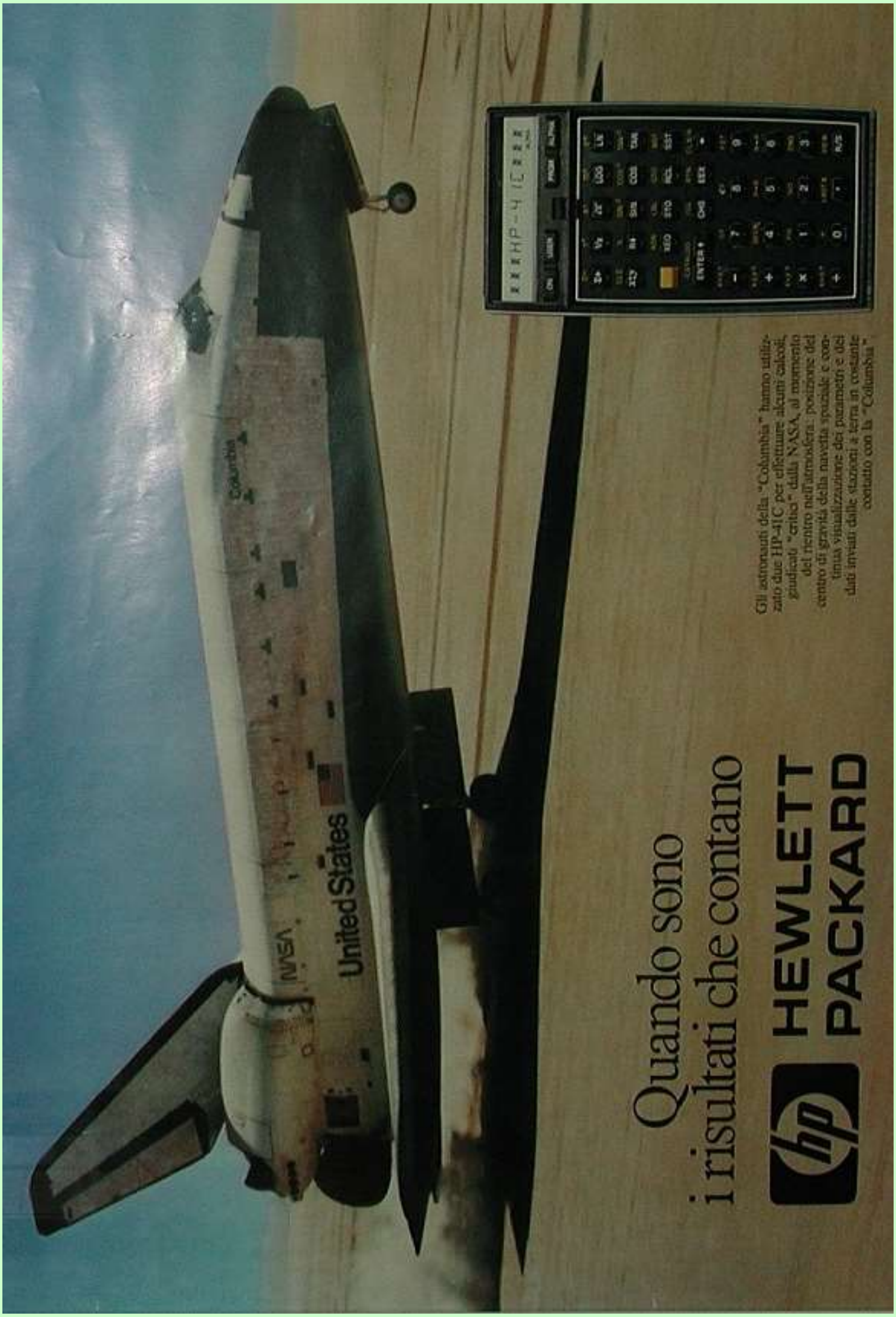
The first fully programmable hand-held calculator, the HP-65 automatically steps through lengthy or repetitive calculations. This advanced instrument relieves the user of the need to remember and execute the correct sequence of keystrokes, using programs recorded 100 steps at a time on tiny magnetic cards. Each program consists of any combination of the calculator's 51 key-stroke functions with branching, logical comparison, and conditional skip instructions.

The HP-65 is priced at $795*. See it, and the rest of the HP family of professional hand-helds at quality department stores or campus bookstores. Call 800-538-7922 (in California, 800-662-9862) for the name of the retailer nearest you.

HEWLETT hp PACKARD

Sales and service from 172 offices in 65 countries

An advertisement of 1975 (above) and another one of 198x (overleaf). Compare the capabilities of the *WP 43S* in your hands. Imagine the opportunities.


Columbia
NASA
United States
HP-41C
Quando sono
i risultati che contano
HEWLETT
PACKARD
Gli astronauti della "Columbia" hanno utilizzato due HP-41C per effettuare alcuni calcoli, giudicati "critici" dalla NASA, al momento del rientro nell'atmosfera: posizione del centro di gravità della navetta spaziale e continua visualizzazione dei parametri e dei dati inviati dalle stazioni a terra in costante contatto con la "Columbia".

SECTION 3: PROGRAMMING

Your *WP 43S* is a powerful *keystroke-programmable* calculator. If already this statement makes you smile with delight, this section is for you. Else we will bring a smile on your face by mentioning the following facts:

Your *WP 43S* allows you to store a sequence of keystrokes like you would use them to solve a problem manually; this is to save you time on repetitive calculations (remember the example on pp. 21ff). Once you have written the keystroke procedure (or *routine*) for solving a particular problem and recorded it in your *WP 43S*, you need no longer devote attention to the individual keystrokes that make up the procedure. You can let your *WP 43S* solve each similar problem for you. And because you can easily check the routine stored, you have more confidence in your final answer since you do not have to worry each time about whether or not you have pressed an incorrect key. Your *WP 43S* performs the drudgery, leaving your mind free for more creative work.

And it becomes even better: You may use program memory for storing more than one routine only. For telling your *WP 43S* where such a routine begins and ends, each one is confined by two steps: it starts with LBL (for LaBeL) and typically ends with RTN (for ReTurN) – cf. p. 23. These two steps separate it from other routines you may add for other tasks. And LBL puts a label on your routine so you can find and call it easily when you want it to be executed.

You may structure program memory even more: Put two or more routines together and separate them by END steps from other routines or sets of routines. What we find between two END steps we call a *program*. Programs are the basic building blocks within program memory. Think of the beginning and end of the entire used program memory section containing implicit END steps.[147] So even with program memory cleared, there will be at least one program within at any time.

Within routines, you may store any sequence of keystrokes (commands, operations, objects). Choose any operation featured – the overwhelming

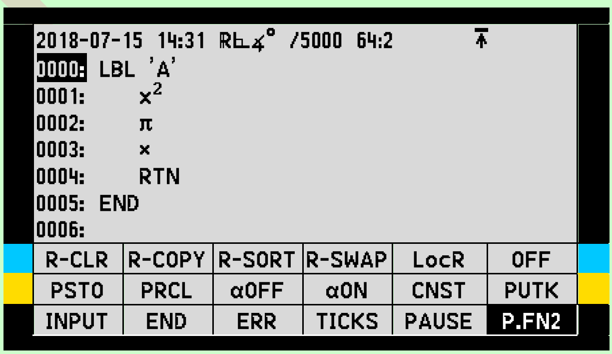
---

[147] You cannot see that first END but the last one is visible – as pictured e.g. overleaf.

majority of them are programmable. The commands in your routine may also access each and every global *register*, variable, or *flag* provided – there are (almost) no limits. You are the sole and undisputed master of the memory. [148]

Each such routine itself may contain one or more *subroutines*. Also subroutines start with LBL and typically end with RTN. Actually, subroutines may look exactly like routines: the only difference is that **a subroutine is called from another routine, while a main routine is called from the keyboard.** Thus we do not need differentiating these two kinds of routines further on.

Enough of theory – press RTN and switch to *PEM* via P/R. The display of your *WP 43S* will change to something like this:



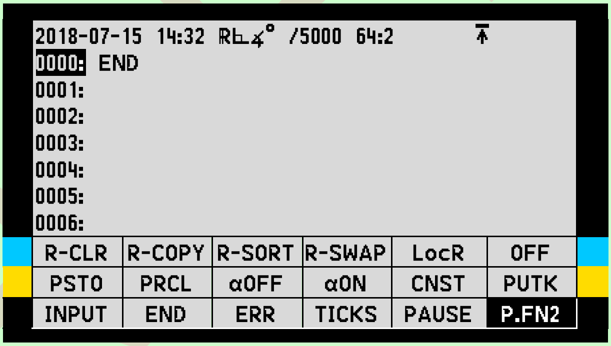
[148] This freedom has a price: Take care that the routines do not interfere with each other in their quest for data storage space. It is good practice to record the global *registers*, variables, and *user flags* a particular routine uses, and to document their purposes and contents for later reference.

An alternative – using *local registers* and *flags* – will be explained further below.

showing the example you entered on pp. 23f (the *status bar* on top will differ according to your time and settings).

In the section of the screen used for numeric output so far, the first seven steps in program memory are shown.[149] Labeled steps and END are 'outdented' for visual structuring. The current position of the *program pointer* (the *current step*) is highlighted by inversion; the routine the program pointer is currently in is called the *current routine*; the corresponding program is the *current program*. The *menu section* displays the top *view* of P.FN.

On the other hand, if you switch to *PEM* for the very first time after unpacking your *WP 43S* (or after resetting it), the display will look like this instead:



Recording a New Routine

Whenever you want to enter a new routine, switch to *PEM* using P/R (unless you are already in) and start with pressing GTO . . . These keystrokes will bring you to the very end of the used section of program memory, so you can start keying in your new routine right there without interfering with anything coded previously.

Start with LBL giving your routine a unique name (it may be up to seven characters long). Then press the keys as you would do in manual

[149] There is no routine-specific step counting like in the *HP-42S* or *HP-35S*.

problem solving (cf. pp. 21ff). Each new step will be inserted right after the *current step*. You find

- LBL for LaBeLing a routine or a program step following,
- XEQ for eXECUting or calling a specific routine,
- RTN for ReTurNing to the caller of the current routine,
- GTO for unconditionally Going TO a specified label (i.e. positioning the program pointer to the respective LBL step),
- R/S for Running or Stopping the current routine,
- ▲ and ▼ (or ≡△ and ≡▽ if there is a *multi-view menu* displayed) for browsing program steps,
- P/R for toggling *Program-entry* and *Run mode*, and
- EXIT for EXITing *PEM* (returning to *run mode*)

all bottom right on your keyboard as shown on p. 203, continued to the left by the *menus* for LOOPs, TESTs, FLAGS, and PARTS. Further programming commands (like END mentioned above) are collected in P.FN. Note that ▲, ▼, ≡△, ≡▽, P/R, and EXIT are not programmable but useful in programming nevertheless (see also p. 211).

Example (from the *HP-15C OH*):

Mother's Kitchen, a canning company, wants to package a ready-to-eat spaghetti mix containing three different cylindrical cans: one of spaghetti sauce, one of grated cheese, and one of meatballs. *Mother's* needs to calculate the base areas, total surface areas, and volumes of the three different cans. It would also like to know, per package, the total base area, surface area, and volume.

Solution:

The program to calculate this information uses these formulas and data:

$$base\ area = \pi\, r^2$$

$$volume = base\ area \times height = \pi\, r^2\, h$$

$$surface\ area = 2\ base\ areas + side\ area = 2\,\pi\, r^2 + 2\,\pi\, r\, h$$

r	***h***	Base Area	Volume	Surface Area
2.5	8.0	?	?	?
4.0	10.5	?	?	?
4.5	4.0	?	?	?
	TOTALS	?	?	?

Method:

1. Enter an ***r*** value into the calculator and save it for other calculations. Calculate the base area ($\pi\, r^2$), store it for later use, and add the base area to a *register* which will hold the sum of all base areas.
2. Enter ***h*** and calculate the volume ($\pi\, r^2\, h$). Add it to a *register* to hold the sum of all volumes.
3. Recall ***r***. Divide the volume by ***r*** and multiply by **2** to yield the side area. Recall the base area, multiply by **2**, and add to the side area to yield the surface area. Sum the surface areas in a *register*.

Do not enter the actual data while writing the program-just provide for their entry. These values will vary and so will be entered before and/or during each program run.

Key in the following program to solve the above problem (assuming *startup default* – and we chose named variables instead of *registers*):

P/R		Switch to *PEM*
GTO . .		
LBL K	LBL 'K'	
STO α ▼ R ENTER↑	STO 'r'	Store radius
x^2	x^2	
π	π	
×	×	Compute base
STO α B A S E ENTER↑	STO 'BASE'	
STO + α g S B ENTER↑	STO+ 'ΣB'	Sum of bases
VIEW VAR BASE	VIEW 'BASE'	Show base for 1 s
P.FN INPUT α ▼ H ENT↑	INPUT 'h'	Enter height

×	×	Compute volume
STO α V O L U M E ENT↑	STO 'VOLUME'	
STO + α g S V ENTER↑	STO+ 'ΣV'	Sum of volumes
VIEW VAR VOLUME	VIEW 'VOLUME'	Show vol. for 1 s
RCL VAR r	RCL 'r'	
/	/	
2	2	Compute side
×	×	
RCL VAR BASE	RCL 'BASE'	
2	2	
×	×	Compute surface
+	+	
STO + α g S S ENTER↑	STO+ 'ΣS'	Sum of surfaces
RTN	RTN	End of routine
P/R		Leave *PEM*

Now, let's run the program:

2.5	2.5_	**1st can:** radius
DISP FIX 2	2.50	
XEQ K	BASE = 19.64	
8	8_	Height
R/S	VOLUME = 157.08	
	164.93	Surface
4	4_	**2nd can:** radius
XEQ K	BASE = 50.27	
10.5	10.5_	Height
R/S	VOLUME = 527.79	
	364.43	Surface
4.5	4.5_	**3rd can:** radius
XEQ K	BASE = 63.62	
4	4_	Height
R/S	VOLUME = 254.47	
	240.33	Surface

RCL VAR ΣB	133.52	**Sum of bases**
RCL VAR ΣV	939.34	**Sum of volumes**
RCL VAR ΣS	769.69	**Sum of surfaces**

> The preceding program illustrates the basic techniques of programming. It also shows how data can be manipulated in *PEM* and *run mode* by entering, storing, and recalling data (input and output) using ENTER↑, STO, RCL, store arithmetic, and programmed I/O. (If you want to run this routine again for another set of cans, remember to clear the variables **ΣB**, **ΣV**, and **ΣS** before.)

See the next paragraphs and the *IOI* for comprehensive information about all the commands used in this example and more.

Labels

As mentioned above, each routine or subroutine begins with a LBL step. Structuring program memory and jumping around within is eased by those labels. You may tag labels not only to the first but to any step in your routine – as known from preceding programmable pocket calculators. Your *WP 43S* allows for specifying a wide variety of alphanumeric labels as described overleaf.

Whenever a step like e.g. GTO ***labl*** is encountered in *run mode* (with ***labl*** representing an arbitrary label), your *WP 43S* will **search this label** using one of the two following methods:

1. If ***labl*** is plain numeric (**00 ... 99**) or **A … J**, it will be searched forward from the current position of the program pointer. When an END step is reached without finding ***labl*** so far, the quest will continue right after previous END (so the search will stay in the *current program*). This is the procedure for *local labels*. So, *local labels* are valid in the *current program* only and may hence be reused in another program.
2. If, however, ***labl*** is an alphanumeric label of up to seven characters of arbitrary case (automatically enclosed in ' like 'Ab1'), searching will

start at program step 0000 and cover the entire program memory (first *RAM,* then *flash memory*) independent of the current position of the program pointer. This is the procedure for *global labels*.[150]

So, *global labels* can be accessed from anywhere in memory, while *local labels* can only be accessed from within their own program.

Addressing labels, on the other hand, follows the rules given below:

1	User input	XEQ, GTO, LBL, LBL?, SOLVE, ∫, f'(x), f"(x), Π_n, or Σ_n		
	Echo	**OP _** (with *TAM* set) e.g. GTO _		
2	User input	α [151] sets *AIM*.	➔ [152] opens indirect addressing.	***local label*** 0 0 … 9 9, A ... J, ***or*** ***1-letter global label*** K, L, T, ..., Z
	Echo	**OP '_**	**OP ➔ _**	**OP *nn*** e.g. LBL 07 or LBL C
3	User input	***Alphanumeric (global) label*** (up to 7 chars[153])	***Stack or lettered register*** X, Y, ..., K	***Register number*** 0 0 … 9 9, . 0 0 … . 9 8[154]
	Echo	**OP '*label*'** e.g. SLV 'F1μ'	**OP ➔ *x*** e.g. ∫fd➔ST.T	**OP ➔ *nn*** e.g. XEQ ➔44

---

[150] These search procedures for local and global labels are as known from the *HP-41C*.

[151] Note you can skip pressing f here – see overleaf. See also an alternative there.

[152] Works with all these operations except LBL .

[153] Said label must contain at least one letter. Labels are case sensitive. The 7th character will terminate entry and close *AIM* – shorter labels need a closing ENTER↑.

[154] ... if the respective *local register* is allocated. Some lettered *registers* may be dedicated to special applications. Check *Addressing and Manipulating Objects in RAM* in *Sect. 1*.

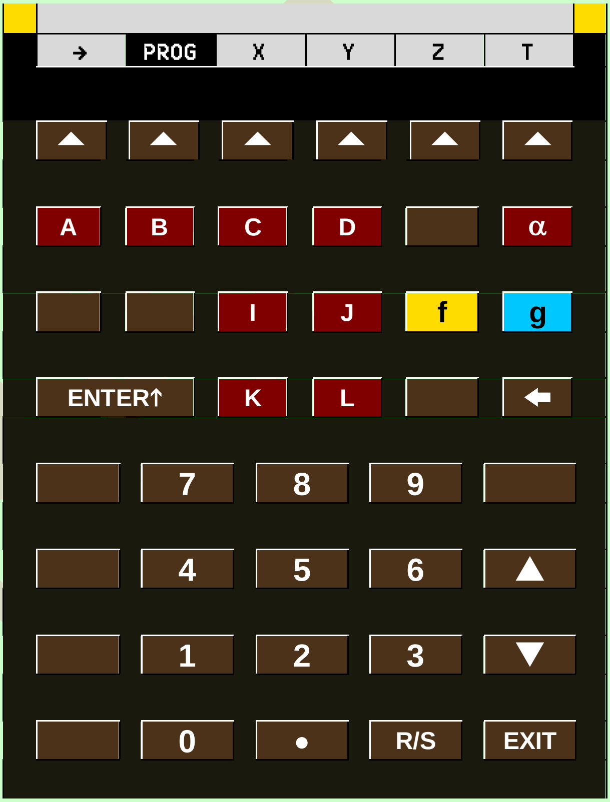
`SLV 'F1µ'` solves the function given in the routine labeled **F1µ** (see pp. 244ff).

`∫fd→ST.T` integrates the function given by PGMINT over the variable whose label is on *stack register* **T** (see pp. 252ff).

`XEQ →44` calls and executes the routine whose label is found in **R44**.

Furthermore, GTO provides two special cases: see GTO**.** and GTO**..** in next chapter.

And remember *TAM* is set during addressing, so the *virtual keyboard* of your *WP 43S* will work like this:



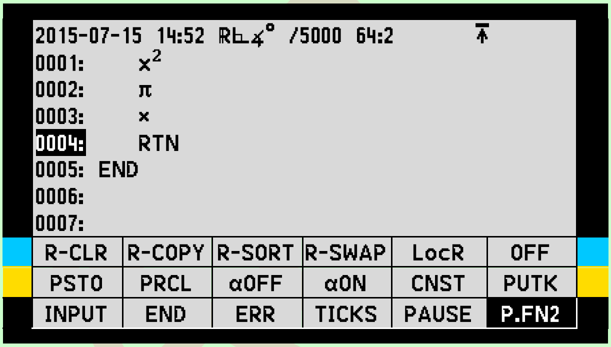
Note you can access the local labels A – D, I, and J directly as well as the global labels K, L, T, and X – Z. This allows reaching up to six programs with only 2 keystrokes.

And note the changed assignment of the 2nd *softkey* compared to p. 57. So, instead of keying in a longer global label in *alpha input mode*, it may be easier to press `PROG` and select it from the *menu* popping up containing all global labels defined at the time of execution.

Editing a Routine

Whenever you want to edit (correct, expand, modify, etc.) an existing routine, start with ensuring you are in *run mode* – then enter GTO ***labl*** . This will position the program pointer onto the corresponding LBL step (as explained on p. 208). Then switch to *PEM* using P/R and start browsing from this LBL step.

Let's browse the program steps in our example routine: press ▼ four times:



Unless you are next to the very beginning of program memory, the program pointer will always be placed in the middle of the *LCD* with three steps displayed above and three steps below of it, if available.

Navigating in program memory, you may execute various actions. If, for example, you want to...

- delete a program step, go to said step (i.e. make it the *current step* by positioning the program pointer on it), then press ←; it will vanish and the program pointer will move on the step before (note this deletion cannot be undone);
- insert something, go to the program step before, and then press the key(s) to be inserted after it;
- continue browsing forward, press ▼ (or ≡▼ if a *multi-view menu* is displayed); when reaching the END, browsing will start with the first step again;

- browse backwards, press ▲ (or ≡△ if a *multi-view menu* is displayed); when reaching program top, browsing will stop;
- go to a particular global label (without inserting a GTO step in the current routine), press GTO . α ***label*** ENTER↑; if you want to go to a local label, press GTO . ***nn*** instead; 1-letter labels can be accessed e.g. via GTO . J.
- start writing a new routine, press GTO . ., then LBL ...

That's almost all. When you are done, press P/R or EXIT to leave *PEM*, returning to *run mode* again.

Running a Routine from the Keyboard (also for Debugging)

Whenever you want to execute an existing routine, ensure you are in *run mode.* Then there are three alternatives:

1. **For normal execution of the current routine:** Press RTN to return the program pointer to the first step of the current routine. Then press R/S. This will run the routine, i.e. start automatically executing the following steps until a STOP, a final RTN, or an END will be encountered[155] where it will halt and display x.
2. **For normal execution of a selected routine:**[156] Press XEQ and specify the label of the program you want to execute (or press PROG and pick the label from the *menu*). This will move the program pointer to the corresponding LBL step (cf. p. 208) and start automatically executing the following steps until a STOP, a final RTN, or an END will be found[155] where it will halt and display x (cf. pp. 21f).
3. **For stepwise execution of a selected routine:** Press GTO instead and specify the label of the program you want to execute (or press PROG and pick it from the *menu*). This will move the program pointer

[155] ... or you interrupt it manually by pressing R/S or EXIT – then it stops after the current step is completely executed. For resuming its execution, press R/S again.

[156] This is the standard way to run routines. Furthermore, you can define shortcuts to your favorite routines by customizing your *WP 43S* as described in *Section 6*.

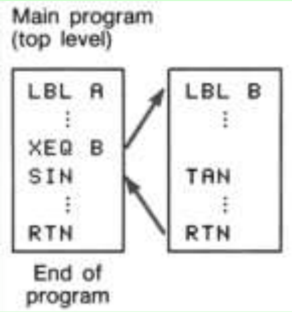
to the corresponding LBL step (cf. p. 208) <u>and wait</u>. Each following program step will then be executed one at a time as you press ▼ (or ≡▽) for it: pressing ▼ will display the step to be executed, releasing it will execute it. When you reach the end of the current routine, ▼ will return to its first step.[157]

Following this procedure, you will go through the routine as in normal execution but significantly slower – and you may perform additional checks after each program step. This procedure is especially useful for *debugging* (i.e. looking for errors in a routine).[158]

If an error occurs while a routine is running, it stops immediately at the step generating said error and throws the corresponding error message (see *App. C* in the *ReM* for a list of all error messages provided). Press any key to clear this *temporary information*; to view the corresponding program step, press P/R.

Subroutines: Running a Routine from another Routine

XEQ is programmable as well. Whenever a running routine encounters an XEQ, it will search for the associated label as described on p. 208, go to it, and continue program execution with the step after this LBL until it encounters a RTN; this will return the program pointer to the step right after above XEQ where execution will continue. Compare the picture where routine **A** calls routine **B**.



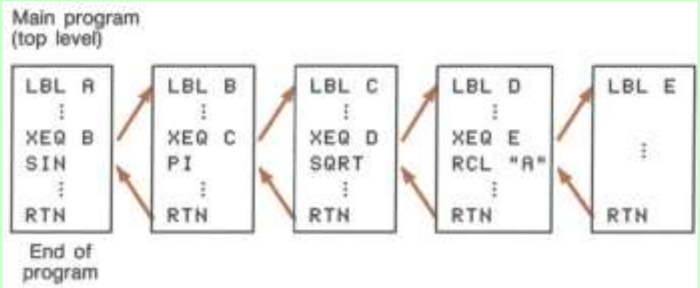
You can also nest subroutines – your *WP 43S* can remember up to eight pending return locations. But all of them will be lost for the current program if you should alter the program pointer while execution of this

---

[157] Pressing ▲ (or ≡△ if a *multi-view menu* is displayed), on the other hand, moves the program pointer backwards in the current routine <u>without</u> executing anything.

[158] Watch that your additional checks, if applicable, do not alter the status of your *WP 43S* in a way deviating from its status in automatic execution; else you shall compensate. Also take care when browsing backwards.

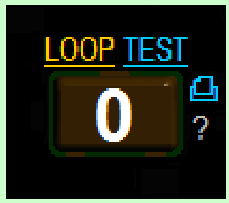
program is stopped; pressing [R/S] or [≡▽] or [▼], however, will not cause a loss of return locations.



If you need any of your subroutines elsewhere, you can call it again at no expense of memory. If you want to call a particular subroutine from another program than the one it is defined in, the label at the beginning of this subroutine must be global.

Automatic Testing and Conditional Branching

So far, we were talking about linear programs running straight from beginning (LBL) to end (final RTN or END). Your *WP 43S* can do more for you: like keystroke-programmable calculators before, it features a set of binary tests checking various calculator states. Most of the binary tests provided are collected in the *menu* TEST. There are also two tests in BITS, and eight tests on *flags* stored in FLAGS. Names of binary test commands contain a '?', most times as their last character.



Generally, binary tests will return `true` or `false` as *temporary information* at left of the **Z** numeric row if called from the keyboard; if called automatically from a routine instead, they will execute the next program step if the test is true at execution time, else skip that step. So the general rule reads "do if true" (or "skip if false").[159] Think of the next step after the test containing a GTO and you see how conditional branching comes into play.

[159] The one and only exception: KEY? skips if true.

Example:

...	
0020: x≤y ?	If this test is true
0021: GTO 'Join'	then go to the label **Join** (at step 32);
0022: x⇄y	else swap ***x*** and ***y***
...	and continue working here.
...	
0032: LBL 'Join'	
0033: ln x	
...	

Most binary tests operate on ***x***. They can **check its *data type***:

- REAL? tests if **X** contains a *real* object (*data type* 2, 8, or 11) and executes the next program step if true, else skips it.
- CPX? tests if **X** contains a *complex* object (*data type* 3 or 9) ...
- MATR? tests if **X** contains a matrix (*data type* 8 or 9) ...
- STRI? tests if **X** contains an *alpha string* (*data type* 7) ...
- SPEC? tests if ***x*** is special (i.e. ±∞ or 'Not a Number') ...
- NaN? tests if ***x*** is 'Not a Number' ...

They can **check its numeric content**:

- INT? tests if ***x*** is an integer number (i.e. has no fractional part) ...
- EVEN? tests if ***x*** is an integer and even ...
- ODD? tests if ***x*** is an integer and odd...
- FP? tests if ***x*** has a nonzero fractional part ...
- PRIME? tests if the absolute value of the integer part of ***x*** is a prime number and executes the next program step if true, else skips it.

They can **compare its numeric content** with 0, 1, or the content of another source specified (let's call it ***s***, cf. also pp. 57 and 60):

- x< ? tests if ***x*** is less than ***s*** and executes the next program step if true, else skips it.

- x≤ ?, x= ?, x≠ ?, x≥ ?, and x> ? work in analogy to x< ?.
- x≈ ? tests if the rounded values of ***x*** and ***s*** are equal and executes the next program step if true, else skips it.

They can **check its internal structure**:

- BC? (or BS?) tests if **X** contains a *short integer*, then checks its bit specified and executes the next program step if said bit is clear (or set), else skips it.
- LEAP? tests if **X** contains a *date*, then extracts the year and tests for a leap year ...
- M.SQR? tests if **X** contains a matrix, then checks if it is square ...

General ***flag*** **tests** operate on the *flag* specified:

- FC? tests this *flag* and executes the next program step if said *flag* is clear, else skips it.
- FC?C works as FC? but clears the *flag* after testing.
- FC?S works as FC? but sets the *flag* after testing.
- FC?F works as FC? but flips the *flag* after testing (i.e. clears it if it was set or sets it if it was clear).
- FS? tests that *flag* and executes the next program step if it is set, else skips it.
- FS?C works as FS? but clears the *flag* after testing.
- FS?S works as FS? but sets the *flag* after testing.
- FS?F works as FS? but flips the *flag* after testing.

Finally, there are **special tests**:

- LBL? tests for the existence of the label specified, anywhere in program memory.
- TOP? will return `true` if the program pointer is in the top level routine (cf. the sketch on p. 213).

- KEY? tests if a key was pressed while a routine was running or paused. If no key was pressed in that interval, the next program step after KEY? will be executed; else it will be skipped and the code of said key will be stored in the address specified. Key codes reflect the rows and columns on the keyboard (see the picture here and p. 224 for an application).

- ENTRY? checks the (internal) entry *flag*. It is set if:
  - any character is entered in *AIM*, or
  - any command is accepted for entry (be it via ENTER↑, a function key, or R/S with a partial command line).

  ENTRY? is useful e.g. after PAUSE.

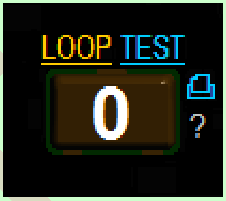
See the *IOI* for more information about all the individual commands contained in TEST, also beyond those mentioned above.

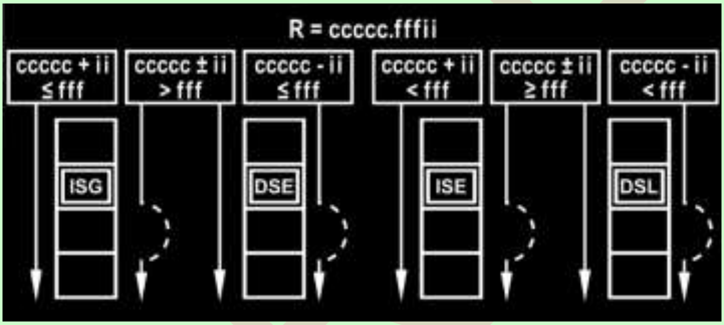
☞ There are further commands also featuring a trailing '?' but returning numbers (e.g. WSIZE?) or codes (e.g. KTYP?) instead of `true` or `false` – you will find these commands in INFO. Turn to the *ReM* for information about them.

As mentioned further above, routines end with RTN (typically) and programs with END. Executing a program, both RTN and END work in a very similar way and show only subtle differences: a RTN immediately after a binary test returning `false` will be skipped – an END will not.

Loops and Counters

The commands DSE, DSL, DSZ, ISE, ISG, and ISZ are for controlling loops in routines. They are all contained in LOOP. Each of them Decrements or Increments a counter in a *register* or variable as specified and executes or skips the following program step depending on the result. See the picture illustrating ISG (Increment and Skip if Greater), DSE (Decrement and Skip if Equal), ISE, and DSL (Decrement and Skip if Less).[160] ISZ and DSZ simply skip if zero. With GTO placed in the skipped step pointing to a label upstream in the same routine you can create loops running until the specified condition is met.





Without such an exit condition you can create an infinite loop – such a routine will run until you interrupt it manually by EXIT or R/S, or until battery voltage falls below the limit. Note that such loops are allowed by the operating system of your *WP 43S*.

Example (also for indirect addressing, cf. *Section 1*):[161]

Write a little routine to store random numbers in **R25** through **R39**.

Solution:

Initialize the loop counter via **25.039** STO 2 4.

Reset the program pointer to the start of program memory by RTN. Switch to programming via P/R and key in:

[160] A similar picture is printed on the back of your *WP 43S*.

[161] This example follows an idea of *Gene Wright*.

LBL X	LBL 'X'	
PROB ▲ RAN#	RAN#	
STO → 2 4	STO →24	
LOOP ISG 2 4	ISG 24	
GTO X	GTO 'X'	if $r24 \leq 39$ return to label **X**
RTN	RTN	else return to *run mode.*
EXIT		

Start this program by pressing XEQ X. It will stop with the last random number in display. Check the target *registers* using RBR.

Example (continued):

Now, write a routine to sort those fifteen stored numbers so the smallest moves to the *register* with the smallest address.

Solution:

We will use the so-called '*bubble sort*' algorithm. Re-initialize the loop counter via **25.039** STO 2 4 (*r24* was modified by program **X** above). Reset the program pointer to the start of program memory by RTN. Switch to programming via P/R and key in:

LBL Y	LBL 'Y'	
P.FN LocR 2 ENT↑	LocR 002	Allocate 2 local *registers*.
LBL A	LBL A	Local label A.
RCL 2 4	RCL 24	Put the start pointer *r24*
STO . 0 0	STO .00	into local *register* 00.
LOOP INC X	INC ST.X	Increment the pointer and
STO . 0 1	STO .01	store it in local *register* 01.
FLAGS CF . 0 0	CF .00	Clear local *flag* 00.
LBL C	LBL C	
RCL → . 0 0	RCL →.00	Recall the contents of the
RCL → . 0 1	RCL →.01	*registers* where *r.00* and *r.01* are pointing to.
TEST x< ? Y	x< ? ST.Y	Is $x < y$?
GTO B	GTO B	Then go to (local) label B
LBL D	LBL D	else…

LOOP ISG . 0 0	ISG .00	increment ***r.00*** and…
ISG . 0 1	ISG .01	if ***r.00*** ≤ 39 increment ***r.01*** and
GTO C	GTO C	if (***r.00*** > 39 or ***r.01*** ≤ 39) return to label C;
FLAGS FS? . 0 0	FS? .00	else check local *flag* 00:
GTO A	GTO A	if set, return to label A,
RTN	RTN	else stop this routine.
LBL B	LBL B	
SF . 0 0	SF .00	Set local *flag* 00.
STO → . 0 0	STO →.00	Store the smaller value
x≷y	x≷y	where ***r.00*** and the greater
STO → . 0 1	STO →.01	where ***r.01*** is pointing to.
x≷y	x≷y	Restore the *stack* and
GTO D	GTO D	return to label D.
EXIT		

Start the program by pressing XEQ Y. Then check the target *registers* using RBR. You will find the smallest value in **R25**, a greater one in **R26**, etc., up to the greatest in **R39**.

Note this program allocates two *local registers* for its exclusive use (**R.00** and **R.01**). Furthermore, it uses 1 local *flag* and 4 local labels.

The following alternative sorting program is even shorter (kudos to *Jean-Marc Baillard* for this routine):

```
LBL 'Z'
   SIGN
LBL A
   RCL L
   RCL L
   RCL →L
LBL B
   RCL →ST.Y
   x> ? ST.Y
   GTO C
   x≷y
   RCL L
   +
```

```
LBL C
   R↓
   ISG ST.Y
   GTO B
   x⇄ →L
   STO →ST.Z
   ISG L
   GTO A
END
```

Just start it by keying in **25.039** XEQ Z .

Cf. *HP-42S OM*, pp. 152 – 154.

Programmed User Interaction and Dialogues

A number of commands are provided for controlling the interaction of programs with you. A program shall output some results to you at least, and it may also ask for your input. In the *IOI*, the behavior of those I/O commands is described if they are entered from the keyboard. Executed by a program, however, they will work differently.

When you start a program by XEQ or R/S, the hour glass ⌛ will start flashing in the *status bar*. While in *manual run mode* each command executed may change the display immediately, in *automatic run mode* only INPUT, PAUSE, STOP, or VIEW will update the display, and this display will hold until the next such command is encountered or *automatic run mode* is left. For programmed I/O, see the following **examples**:

- Take VIEW for displaying intermediate results. Specify any *register* or variable you want as source of information – also **X** is a valid parameter of VIEW. The name of the source will label the output.

  ☞ Frequent display updates will slow down program execution, since the anti-flicker logic waits for a complete display refresh cycle before allowing the next update.

- Use

`VIEW xyz` `PAUSE nn`	for displaying output <u>for a defined minimum time interval</u>, specified by PAUSE.

- If you have a printer connected, you may send your program output thereto as well. Turn to pp. 233ff for more about printing.

- Ask ('prompt') for <u>numeric input</u> employing

`VIEW xyz`	update display showing the *register* or variable ***xyz***,
`STOP`	... and wait for user reaction, finished by R/S.
`STO xyz`	store what the user entered.

☞ A stop sign will be displayed in the *status bar* when the program pointer runs on STOP. Whatever you will key in will be put into ***xyz*** when you continue program execution by R/S.

More elegant is using INPUT for this task:

`INPUT xyz`	does the same in just one step.

- Prompt for <u>alphanumeric input</u> using the following steps:

`SF ALPHA`	sets *AIM* for upcoming input.
`RCL xyz`	displays the *register* with the message string.
`STOP`	waits for your input. Whatever you key in now is appended to ***x*** when you continue by pressing R/S.
`CF ALPHA`	returns to the numeric mode previously set.
`STO xyz`	stores ***x*** to wherever you like.

Again, more elegant is using INPUT for this task:

`SF ALPHA`	sets *AIM* for upcoming input.
`INPUT xyz`	
`CF ALPHA`	returns to the numeric mode previously set.

- If you need to enter values for several variables then the following way is most efficient (although it may look lengthy here):

`LBL 'Var.In'`	we will need this label for VarMNU later.
`MVAR 'xy1'`	
`MVAR 'xy2'`	
`MVAR 'xy3'`	
`...`	
`VarMNU 'Var.In'`	creates a *menu* for the variables defined immediately after 'Var.In' and shows it.
`STOP`	stops for user interaction.
`EXITall`	exits the *menu* when program continues.
`RCL 'xy2'`	recalls what you need first (it may have been entered in any order).

The label called '**Var.In**' here should be located close to the program top. It shall be followed by up to 18 MVAR steps defining your variables required. When the program encounters the step VarMNU, it will setup a *menu* for these variables and display it. Here, this would look like

Now, if you want to

- **write** a new value into one of the variables displayed, key in the value or calculate it, then press the corresponding *softkey*. The content of **X** will be stored.
- **recall** the present value of one of the variables displayed, enter RCL,[162] then press the corresponding *softkey*.
- **view** the present value of one of the variables displayed, enter VIEW, then press the corresponding *softkey*.[163]
- **exit** this *menu*, press EXIT.
- **continue** program execution, press R/S.

---

[162] The standard *menu* as shown on p. 57 will <u>not</u> appear after RCL here.

[163] The standard *menu* as shown on p. 56 will not appear after VIEW here. Note that the *HP-42S* allowed for just viewing the present value of one of the variables displayed by pressing f and the corresponding *softkey*; we cannot support this on your *WP 43S* since it offers you three *menu* rows.

- Directly react on particular keys pressed: The key codes returned by KEY? (cf. p. 217) allow for 'real time' response to user input from the keyboard. KEY? takes a *register* argument (**X** is allowed but does not lift the stack) and stores the key most recently pressed during program execution or pause in the *register* specified.[164] Although the keyboard is active during program execution it is desirable to display a message and suspend the routine by PAUSE while waiting for user input. Since PAUSE will be terminated early by a key press, simply use PAUSE **99** in a loop to wait for input. Since KEY? acts as a test as well, a typical user input loop may well look like this:

LBL 'US.in'	
RCL *xyz*	displays the *register* with the message string.
PAUSE 99	waits 9.9 s for user input unless a key is pressed.
KEY? 00	tests for user input and puts the key code in **R00**.
GTO 'US.in'	If there was no input then return to the beginning;
LBL? →00	else: if a label corresponding to the key code exists…
XEQ →00	… then call it, …
GTO 'US.in'	… else return to the beginning.

Instead of the dumb waiting loop, the routine can do some computations and update the display before the next call to PAUSE and KEY?

To be even more versatile, you can use KTYP? to return the type of the key pressed if its row / column code is given (see the *IOP*).

If you decide not to handle the key in your program you may feed it back to the main processing loop of the *WP 43S* with the command PUTK ***nn*** . It will cause the program to halt, and the key will be handled as if pressed after the stop. This is especially useful if you want to allow numeric input while waiting for some special keys like the arrows. After execution of the PUTK command you are responsible for letting the routine continue its work by pressing R/S.

See the *IOI* for more information about the commands mentioned in this chapter and their parameters.

---

[164] Note R/S and EXIT cannot be queried since they stop program execution immediately.

Solving Differential Equations

The following method uses the programmability of your *WP 43S* for solving ordinary 2nd order differential equations, a type frequently occurring in physics. [165]

In a **first example**, we will solve the equation of motion for the fall of a parachutist $\frac{d^2 f}{dt^2} = g - b\left(\frac{df}{dt}\right)^2$ with earth acceleration g and b taking care of drag.

Proceeding in small constant time steps *Δt*, the following set of equations controls the vertical motion of the parachutist (or skydiver):

$$\left(\frac{df}{dt}\right)_{1/2} = \left(\frac{df}{dt}\right)_0 + \left[g - b\left(\frac{df}{dt}\right)_0^2\right] \times \frac{\Delta t}{2}$$

$$f_1 = f_0 + \left(\frac{df}{dt}\right)_{1/2} \quad \text{and} \quad \left(\frac{df}{dt}\right)_{3/2} = \left(\frac{df}{dt}\right)_{1/2} + \left[g - b\left(\frac{df}{dt}\right)_{1/2}^2\right] \times \Delta t,$$ [166]

$$f_2 = f_1 + \left(\frac{df}{dt}\right)_{3/2} \Delta t \quad \text{etc.}$$

Assume start height at time zero ($t = 0$) is 1000 m and vertical velocity is zero (i.e. $f(t=0) = f(t_0) = f_0 = 1000$ and $\left(\frac{df}{dt}\right)_0 = 0$). Using named variables ***Δt***, ***b***, ***t***, ***f***, and ***df/dt***, the following routine will compute height and velocity of the parachutist as functions of time:

```
LBL 'PFall'
   .5              initialize all variables used
   STO 'Δt'
   .003            assumed realistic drag value for a falling body
   STO 'b'
   1000            start height
   STO 'f'
```

---

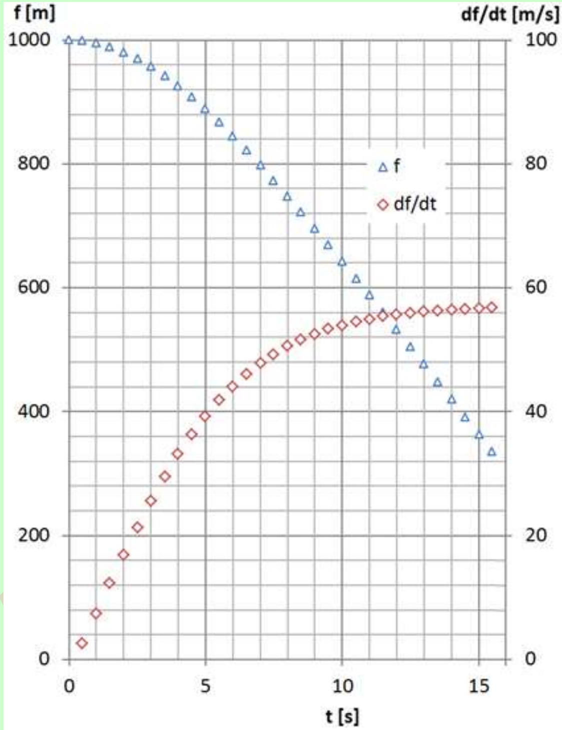
[165] Turn to the *ReM*, *App. H*, for background information about the method applied here.

[166] Note the contents of the rectangular brackets must be ≥0 always. Thus, this routine will work for velocities $< \sqrt{g/b}$ only, not for abruptly decelerating fast initial movements (e.g. by opening a parachute).

0	start time and velocity	
STO 't'		
STO 'df/dt'	end of initialization	
LBL 01	**begin of time loop**	
# $g_\oplus$	take $\mathbf{g}_\oplus$ out of <u>CONST</u>	
RCL 'b'		
RCL 'df/dt'		
x^2		
×	**b** × (**df/dt**)^2	
-	**g** – **b** × (**df/dt**)^2	
RCL 't'		
x>0 ?	check **t**ime – it will be zero in 1st run	
GTO 02	from 2nd run on go to local label **02**	
DROP	1st run only: forget **t**	
2	1st run only:	
/	1st run only: [**g** – **b** × (**df/dt**)^2] / 2	
GTO 03	1st run only: go to common part	
LBL 02	from 2nd run on:	
DROP	from 2nd run on: forget **t**	
LBL 03	common part of time loop resumes here again	
RCL× 'Δt'	[**g** – **b** × (**df/dt**)^2] × **Δt** (or half of it in 1st run)	
STO+ 'df/dt'	calculate the new **df/dt**	
RCL 'Δt'		
STO+ 't'	calculate the new **t**ime	
RCL× 'df/dt'	**df/dt** × **Δt**	
STO- 'f'	calculate the new **f**	
VIEW 't'	display new time	for plotting next data points (**t** ; **f**) and (**t** ; **df/dt**) .
STOP		
VIEW 'f'	display new height	
STOP		
VIEW 'df/dt'	display new velocity	
STOP		
GTO 01	**end of time loop, return for next run through it**	
END		

Now, leave *PEM* and start program execution via XEQ PROG PFall – plotting the points calculated will result in a diagram like the one overleaf. Height decreases following a parabola over time in the beginning but becomes linear later. Note the vertical velocity does not increase much anymore after some 12 s here, approaching some 57 m/s while skydiving with closed parachute.

For comparison: the velocity limit with an open parachute ($b = 0.3$) will be < 6 m/s, so the vertical velocity at touchdown will be like falling from a wall 1.65 m high.



In a **second example**, we will demonstrate solving a 2D problem like e.g. finding the orbit of a satellite in the gravitational field of the earth. Here we have a pair of coupled differential equations. This problem is solved as follows:

$$\left(\frac{dx}{dt}\right)_{1/2} \approx \left(\frac{dx}{dt}\right)_0 + K_x \frac{\Delta t}{2} \qquad \left(\frac{dy}{dt}\right)_{1/2} \approx \left(\frac{dy}{dt}\right)_0 + K_y \frac{\Delta t}{2}$$

$$\left(\frac{dx}{dt}\right)_{i+\frac{1}{2}} \approx \left(\frac{dx}{dt}\right)_{i-\frac{1}{2}} + K_x \Delta t \qquad \left(\frac{dy}{dt}\right)_{i+\frac{1}{2}} \approx \left(\frac{dy}{dt}\right)_{i-\frac{1}{2}} + K_y \Delta t$$

$$x_{i+1} \approx x_i + \left(\frac{dx}{dt}\right)_{i+1/2} \Delta t \qquad y_{i+1} \approx y_i + \left(\frac{dy}{dt}\right)_{i+1/2} \Delta t$$

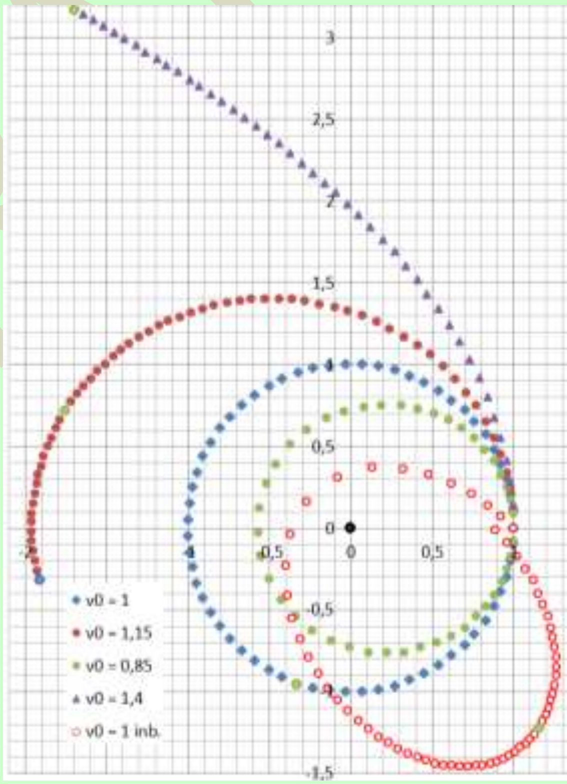
with $-\frac{GM}{(x^2+y^2)^{3/2}}\, x = K_x$ and $-\frac{GM}{(x^2+y^2)^{3/2}}\, y = K_y$.

So, here is some crosstalk (a.k.a. coupling) between ***x*** and ***y***. Nevertheless, proceeding like we did in the first example above, the following routine will compute the coordinates ***x*** and ***y*** of the satellite as functions of time. For ease of handling in a first calculation, we set $GM = 1 = a$

and the start values $x_0 = 1$, $\left(\frac{dx}{dt}\right)_0 = 0$, $y_0 = 0$, $\left(\frac{dy}{dt}\right)_0 = 1$. These 'variable' start values shall be entered using INPUT here (cf. p. 222):

LBL 'Satell'	
INPUT 'x'	start of variable initialization
INPUT 'y'	
INPUT 'dx/dt'	
INPUT 'dy/dt'	
.1	initialize the remaining 'fixed' start values
STO 'Δt'	
1	(for earth satellites, take GM out of <u>CONST</u> instead)
STO 'a'	
0	start at time zero
STO 't'	end of initialization
LBL 01	**begin of time loop**
RCL 'y'	
RCL 'y'	
x^2	y^2
RCL 'x'	
x^2	
+	y^2+x^2
-1.5	
y^x	$(y^2+x^2)^{-1.5}$
RCL× 'a'	$a\,(y^2+x^2)^{-1.5}$
×	$y\,a\,(y^2+x^2)^{-1.5} = -K_y$
RCL L	$a\,(y^2+x^2)^{-1.5}$
RCL× 'x'	$x\,a\,(y^2+x^2)^{-1.5} = -K_x$. *Stack* is $[-K_x, -K_y, ...]$ now.
RCL 't'	
x>0 ?	check **t**ime – it will be zero in 1st run
GTO 02	from 2nd run on go to local label **02**
DROP	1st run only: forget **t**
2	1st run only:
/	1st run only: $-K_x/2$
x⥂y	1st run only: *stack* is $[-K_y, -K_x/2, ...]$ after **x⥂y**
2	1st run only:
/	1st run only: $-K_y/2$
x⥂y	1st run only: *stack* is $[-K_x/2, -K_y/2, ...]$ after **x⥂y**
GTO 03	1st run only: go to common part of time loop
LBL 02	from 2nd run on:
DROP	from 2nd run on: forget **t**

`LBL 03`	the common part of the time loop resumes here again
`RCL× 'Δt'`	− K_x × **Δt** (or half of it in 1st run)
`STO- 'dx/dt'`	calculate the new **dx/dt**
`DROP`	bring ***y*** to the front
`RCL× 'Δt'`	− K_y × **Δt** (or half of it in 1st run)
`STO- 'dy/dt'`	calculate the new **dy/dt**
`RCL 'Δt'`	
`STO+ 't'`	calculate the new **t**ime
`RCL× 'dx/dt'`	**dx/dt** × **Δt**
`STO+ 'x'`	calculate the new **x**
`VIEW 'x'`	display the new **x** for plotting
`STOP`	
`RCL 'Δt'`	
`RCL× 'dy/dt'`	**dy/dt** × **Δt**
`STO+ 'y'`	calculate the new **y**
`VIEW 'y'`	display also the new **y** for plotting the new point (**x**, **y**)
`STOP`	
`GTO 01`	**end of time loop, return for next run through it**
`END`	



Plotting the points calculated for these start values will result in a perfect circle as shown by the blue symbols in the diagram – taking 64 time steps for one orbit. We added some examples with slightly different start velocities for comparison. The green elliptical orbit takes 46 **Δt** only, the dark red one 116. Green and blue marks in the other curves highlight the positions after 46 and 64 **Δt** for comparison.

The innermost red ellipse starts with velocity 1 again but directed 45° inwards ('NW'). Note this curve does not close due to the perigee speed being too high for the time step **Δt** chosen. We recommend watching the limits of such numeric models always.

For a descent to a planet or a moon, you can introduce a decelerating force which may even depend on height over ground. Your imagination is the limit – and your ability in mathematical modeling.

The Programmable Menu (MENU)

Your *WP 43S* has a *programmable menu* which is used to cause program branching. By this, you can create *menu*-driven programs. The MENU function selects the *programmable menu*. The *menu* is displayed when the program stops. You can define each *softkey* in this *menu* so that when this key is pressed, a particular GTO or XEQ instruction will be executed. You can even re-define ▲, ▼, and EXIT.[167]

To define a *softkey* in the *programmable menu*:

1. Store a string of up to seven characters in *register* **K**. This is the text that will appear in the *menu space* for the *softkey* specified (If space does not suffice, only the first characters will be displayed). **K** is not used when defining ▲, ▼, or EXIT.

2. Call KEYG (i.e. on key, go to) or KEYX (i.e. on key, execute). You find them in P.FN.

3. Specify which key you want to define (the *menu view* changes):

   a. Press one of the 18 *softkeys* available, ▲, ▼, or EXIT.

   b. Alternatively, enter the respective key number, 1 through 21 (unshifted leftmost *softkey* carries #1, g-shifted leftmost #13).

[167] See *HP-42S Programming Examples and Techniques*, pp. 29 - 39, 92 - 99, 158 - 160, and 184 - 192, for some sample programs using the *programmable menu*.

4. Specify a program label using one of the following three methods (the *menu view* changes to the one shown on p. 210):
   a. Select an existing global label by pressing the corresponding *softkey* in PROG.
   b. Key in a global label character by character using *AIM*.
   c. Key in a two-digit local label.

Repeat this procedure for each *softkey* in the *programmable menu* you want to define. The new definition replaces any previous definition that may exist for that *softkey*.

To display the *programmable menu*:

Execute the MENU function, e.g. by entering P.FN P.FN2 MENU.

To clear all *softkey* definitions in the *programmable menu*:

Call CLMENU (clear the *programmable menu*), e.g. by entering CLR CLMENU.

Basic Kinds of Program Steps

You have seen various program steps so far. Each step takes a single place in program memory, and each step is numbered automatically. Basically, the contents of these steps fall into four categories – one program step may contain...

- a global or local **label** (like LBL 'Join' or LBL 07 above) or
- a complete **command** (like - or y^x or STO× →'Prd2') or
- an entire ***alpha string*** (a.k.a. an *alphanumeric constant*, like "This is a text."; such a text will be automatically stored in **K**) or
- an entire **number** (a.k.a. a *numeric constant*, like -1.902×10^{-16} or $1\,23\,45_{16}$ or # λ_c; such a constant will be automatically stored in **X**).

Since each constant takes one step, there is no need for separating them by ENTER↑ in a routine.

Example:

Think of calculating 12.3 + 45.67 in a routine.

Then pressing **12.3** ENTER↑ ◀ **45.67** + will result in a program snippet

```
12.3
45.67
+
```

which will do for returning 57.97. The missing ENTER↑ saves two bytes of program space and makes the routine a tiny bit faster. You will achieve the same by **12.3** EXIT **45.67** +. It may not be really important here but you should know.

Constant vectors and matrices cannot be entered directly in a program; though you can store them in *registers* or variables and manipulate these stored *items* (as described in *Section 2*) in routines as well.

Program steps may require two or more bytes of memory. We think you will hardly ever run out of program space (but you may, of course: if you do while trying to enter a new program step, you will read an error message RAM is full ; see *App. B* of the *ReM* for ways to escape from such a situation).

Deleting Programs

To delete some steps of a program, proceed as explained on pp. 211f. Repeat as often as necessary.

To delete an entire program, move the program pointer into this program (e.g. by entering GTO . and picking the label of this program) and then press CLR CLP ENTER↑. Note CLP will remove the entire program from memory, not only the routine the program pointer is in. And CLP cannot be undone! The space freed by CLP will be added to the pool of free space your *WP 43S* features.

To delete <u>all</u> programs stored in RAM, press CLR CLPall and confirm. Thereafter, program memory will be completely wiped out. Note that also CLPALL cannot be undone.

Serial Input and Output of Data and Programs

Xxx

Local Data

After some time with your *WP 43S* you will have a number of routines stored, so keeping track of their resource requirements may become a challenge. Most modern programming languages take care of this by declaring *local variables*, i.e. memory space allocated from general data memory and accessible for the *current routine* only; when the routine is finished, the respective memory is released. On your *WP 43S*, mainly *registers* are used for data storage – so we offer *local registers* to you allocated to your routines exclusively.

Example:

Let's assume you write a routine labeled **P1** and need five *registers* for your computations therein. Then all you have to do is just enter *PEM*, go into the routine **P1**, and enter

P.FN LocR 5 ENTER↑

specifying that you want five *local registers*. Thereafter, you can access these *registers* by using local addresses **.00** … **.04** throughout **P1**.

Now, if you call another routine **P2** from **P1**, also **P2** may contain a step LOCR, requesting *local registers* again. These will also carry *local register* addresses **.00** etc., but the *local register* **.00** of **P2** will be physically different from the *local register* **.00** of **P1**, so no interference will occur. As soon as the return step is executed, the *local registers* of the corresponding routine are released and the space they took is returned to the pool of free memory.

In addition, you get sixteen local *user flags* as soon as you request at least one *local register*.

Local data holding allows for recursive programs, since every time such a routine is called again it will allocate a new set of *local registers* and *user flags* being different from the ones it got before. See the commands LOCR, LOCR?, MEM?, and POPLR in the *IOI*; and look up *App. B* of the *ReM* for more information, also about the limitations applying to local data.

Flash Memory (*FM*)

In addition to the *RAM* provided, your *WP 43S* allows you to access *FM* for voltage-fail-safe storage of user programs and data. The first section of *FM* is a *backup region*, holding the image of the entire *RAM* (i.e. user program memory, *registers*, and *WP 43S* states) as soon as you have executed a SAVE. The remaining part of *FM* is for programs only.

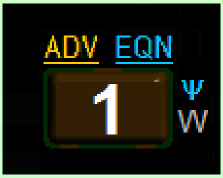
Global labels in *FM* can be called using XEQ like in *RAM*. This allows creating program libraries in *FM*. Use CATALOG'PROGS'FLASH to see the global labels already defined in *FM*.

FM is ideal for backups or other relatively long-living data, but shall not be used for repeated temporary storage like in programmed loops.[168] Conversely, *registers* and standard user program memory residing in *RAM* are designed for data changing frequently but will not hold data with the batteries removed for longer than a few minutes. So both *RAM* and *FM* have their specific advantages and disadvantages you should take into account for optimum benefit and longevity of your *WP 43S*.

[168] *FM* may not survive more than some 100 000 flashes. Thus, we made commands writing to *FM* (SAVE or PSTO) non-programmable.

SECTION 4: ADVANCED PROBLEM SOLVING

There are some powerful commands provided for computing programmable sums and products, for solving equations, for computing



definite integrals as well as 1st and 2nd derivatives. All are contained in ADV or EQN. Pressing ADV in *run mode* results in

The commands Σ, Π, SLVQ, SOLVE, ∫, f′(x), and f″(x) are explained below in this order. All these commands may also be programmed. Integrating, deriving, and solving equations interactively may be reached through EQN. See below for details and examples.

Programmable Sums

The command Σ is called with a loop control number in **X** and a label trailing the command. Said loop control number follows the format `ccccc.fffii` (as it does in DSE etc. mentioned above).

In its heart, Σ then works like this:

1. It sets the sum to **0** initially.

2. It fills all *stack registers* with `ccccc` and calls the routine specified by the label. That routine returns a summand in **X**.

3. It adds this summand to said sum.

4. It decrements `ccccc` by `ii`; if `ccccc` ≥ `fff` then Σ goes back to step 2, else it returns the final sum in **X**.

   If `ii` = 0 , `ccccc` will be decremented by **1** in each loop.

Example:

Compute $\sum_{k=0}^{100} \sqrt{k}$

Solution:

1. Write a little program for the internal calculation of the summands:

```
LBL 'ΣSQRT'
   √x
   RTN
```
.

2. Enter
   100
   ADV Σn α g S S Q R T ENTER↑
   (or pick ΣSQRT from PROG, cf. p. 210)
   and get 671.462 9 returned if FIX **4** is set.

Σ deliberately sums from the last term to the first, on the assumption that summations will often be of convergent series and this summing order should generally increase accuracy.

Programmable Products

The command Π is called with a loop control number in **X** and a label trailing the command (like for the command Σ).

In its heart, Π works almost as Σ :

1. It sets the product to **1** initially.
2. It fills all *stack registers* with `ccccc` and calls the routine specified by the label. That routine returns a factor in **X**.
3. It multiplies this factor with said product.
4. It decrements `ccccc` by `ii`; if `ccccc` ≥ `fff` then Π goes back to step 2, else it returns the final product in **X**.

   If `ii` = 0 , `ccccc` will be decremented by **1** in each loop.

Example:

Compute $\prod_{k=1}^{50} \frac{1}{\sqrt{k}}$

Solution:

1. Write a little program for the internal calculation of the factors:

```
LBL 'PROD'
   √x
   1/x
   RTN
```
.

2. Enter
   50.001
   ADV Π_n α P R O D ENTER↑
   (or pick PROD from PROG, cf. p. 210)
   and get 5.734×10^{-33} returned if SCI **3** is set.

Solving Quadratic Equations

The command SLVQ finds the *real* and *complex* roots of a quadratic equation $ax^2 + bx + c = 0$ with its *real* parameters on the input *stack* [***c***, ***b***, ***a***, …] :

- If $r := b^2 - 4ac \geq 0$, SLVQ returns the 2 *real* roots $-\frac{b \pm \sqrt{r}}{2a}$ in **Y** and **X**. If called in a routine, the step after SLVQ will be executed then.
- Else, SLVQ returns the 1st *complex* root in **X** and the 2nd in **Y** (the *complex conjugate* of the 1st). If called in a routine, the step after SLVQ will be skipped then.

So actually, SLVQ tests for *real* roots at its very end. In either case, SLVQ returns ***r*** in **Z**. Higher *stack registers* are kept unchanged. **L** will contain equation parameter ***c***.

Example:

Find the roots of $4x^2 - 3x - 2 = 0$.

Solution:

4 ENTER↑ **3** +/– ENTER↑ **2** +/–

ADV SLVQ returns (with FIX **4** chosen) x = **1.175 4**, y = **– 0.425 4**, z = **41.000 0**. Since z is positive, x and y are the two *real* roots of this equation here.

Check:

Store x in **J** and y in **K**. Then enter

RCL J FILL **4** × **3** − × **2** − returning 2.000 0×10^{-33} and

RCL K FILL **4** × **3** − × **2** − returning 0.000 0.

Remember your *WP 43S* calculates with 34 digits precision, so any result within $\pm 3 \cdot 10^{-33}$ is equal to zero in this matter.

General Equations

The *menu* EQN lets you store, select, and edit arbitrary equations; you may use each such equation for

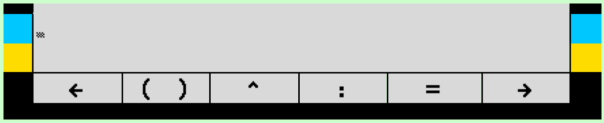
- solving it interactively for any variable it contains, for
- integrating and
- deriving.

The number of equations you can store and the number of variables used in each equation are limited only by the amount of free memory available.

Example:

Press EQN. If there are no equations in memory yet, your *WP 43S* will return:

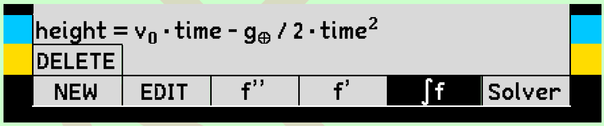
Press NEW to enter a new equation. You will get immediately:



with *alpha input mode* turned on (cf. pp. 189ff). Enter your equation now, e.g.

▼ height = v R↓ 0 × time − g⊕ / 2 × time ^ 2 [169]

for the height of e.g. a ball thrown vertically upwards with velocity v_0. Press ENTER↑ for closing the *Equation Editor* and see: [170]



You will get such a display whenever one or more equations are stored. The equation shown is called the *current equation*. (A dashed line will show up when there are more equations; to select another one as the current equation, press ▲ or ▼ until the requested equation appears.)

Pressing EDIT opens the *Equation Editor* for the current equation:

You may modify this equation at any position by moving the edit cursor to the location behind the character(s) to be corrected and pressing ⌫ followed by the new character(s) to be inserted here.

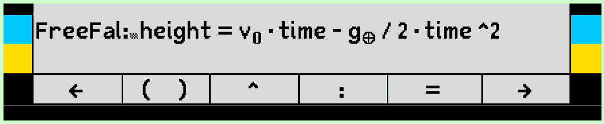
For labeling this equation, move the cursor left to its very begin using ←, and key in:

---

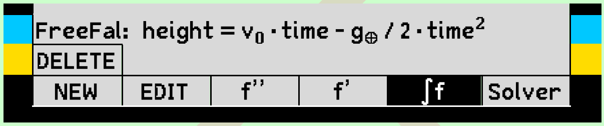
[169] The index of the earth acceleration constant is found in the punctuation *menu* α•, reached via g . in AIM.

[170] Note MULT· is set here for sake of better readability of equations. And some spaces are inserted automatically for the same reason.

F ▼ R E E ▲ F ▼ A L :



ENTER↑

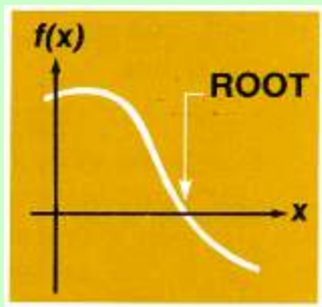


ENTER↑ closes the *Equation Editor* storing the modifications you made. Note editing an equation clears all its variables.

If an equation become wider than the display ellipses will be displayed at its end(s); then use ← and → in the *Equation Editor* for scrolling.

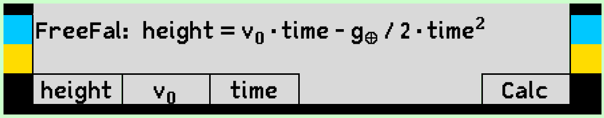
The Interactive Solver for Arbitrary Equations

The built-in *Solver* application of your *WP 43S* is a special root finder that enables you to solve an equation for any of its variables. It allows for solving for an arbitrary unknown as well as for finding the root(s) of an arbitrary equation.[171]



Press EQN, make the equation you want to solve the *current equation* (see previous chapter), and press Solver. Your *WP 43S* will check this equation for syntax errors (missing operators, misspelled functions, illegal variable names, etc.). It will then return a *menu* of all applicable variables, like the one in our **example**:

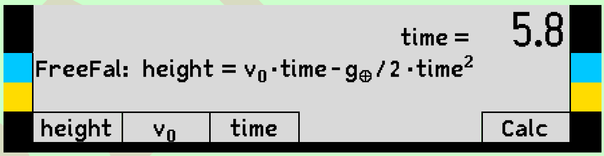
[171] Translator's note for German readers: Der eingebaute Gleichungslöser („Solver") Ihres *WP 43S* erlaubt Ihnen das Auflösen nach einer beliebigen Unbekannten bzw. das Finden der Nullstellen einer beliebigen Gleichung.



Note your *WP 43S* knows **g⊕** is a constant contained in CONST. Now you can enter values for any variables you know by pressing the respective *softkeys*, e.g. **–50** **height** **20** **v_0** (corresponding to 50 *m* below start height and a velocity of 20 *m/s* upwards at time zero), until only one variable remains unknown. Optionally, enter one or two initial guesses for the unknown like **5** **time** **10** **time**. Set the display format and precision unless done before:

DISP FIX 0 1 .

Now, press the *softkey* for the unknown **time** once more (but now without any numeric input heading), and your *WP 43S* will solve the equation for this variable and return its value in **X**:



corresponding to 5.8 *s* until your ball passes said point. Note entering the known values and guesses disabled *automatic stack lifting*.

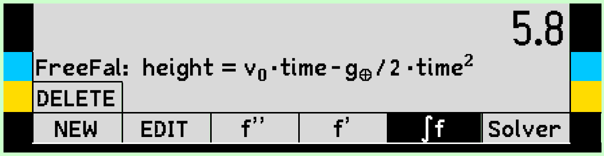
Another **example** (from the *HP-27S OM*):

Carbon-14 Dating. Wood on the outer surface of a giant sequoia tree exchanges carbon with its environment. The radioactivity of this wood is 15.3 counts per minute per gram of carbon. A sample of wood from the center of the tree yields 10.9 counts per minute per gram of carbon. The rate constant for the radioactive form of carbon, ^{14}C, is 1.20×10^{-4}. How old is the tree? What is the half-life of ^{14}C?

Solution (assuming you continue directly after previous example):

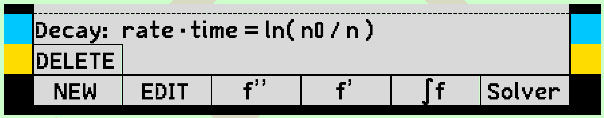
Exit the current equation and enter a new one for radioactive decay:

EXIT

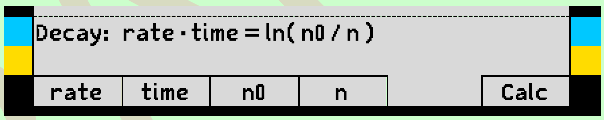


NEW

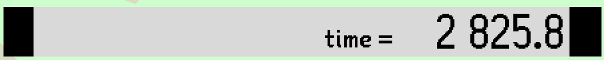
D ▼ E C A Y : R A T E × T I M E = L N ( ) ← N 0 / N
ENTER↑



Solver



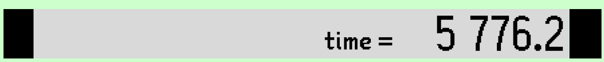
1.2 E +/- **4** rate **15.3** n0 **10.9** n time



This is the computed age of this tree in years.

Now, calculate the half-life of ^{14}C, that is the time required for half the material present to decay:

2 n0 **1** n time



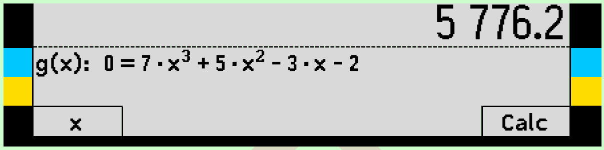
Good guess! Meanwhile, half-life of ^{14}C is known to be 5 730 ± 40 years.

One more **example:**

Find the roots of $7x^3 + 5x^2 - 3x - 2 = 0$.

Solution:

1. Enter the equation as demonstrated above.
2. Make this equation the *current equation* and press [Solver]. You will see:



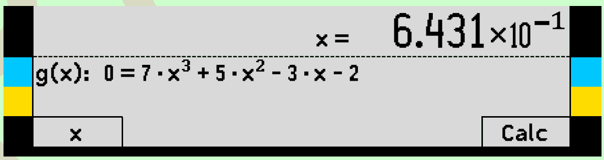
3. Optionally enter one or two initial guesses for the unknown like

   0 [x] **1** [x] .

4. Set the display format and precision

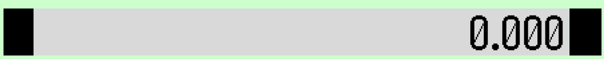
   [DISP] SCI [3]

5. Press the *softkey* [x] for the unknown once more (but now without any numeric input heading), and your *WP 43S* will solve the equation for this variable and return its value in **X**:



6. If you want to crosscheck you can enter

   [RCL] [x] [Calc] returning



   confirming the result of the *Solver*. Slightly greater ***x***-values, e.g.

   .65 [x] [Calc], return positive values for ***g***(*x*), while slightly smaller values, e.g.

   .64 [x] [Calc], return negative values for ***g***(*x*).

7. There may be one or two more roots:

a. Enter two new initial guesses for the unknown like **–2** [x] **0** [x]. Press the *softkey* for the unknown once more, and you will get:

x = -8.064×10^{-1}

Slightly greater ***x***-values, e.g. **–0.8**, return positive values for ***g***(x), slightly smaller values, e.g. **–0.81**, return negative values for ***g***(x). Thus, there must be one more root between the two roots found.

b. Enter two new initial guesses for the unknown like **–.7** [x] **.5** [x]. Press the *softkey* for ***x*** once more and you will get:

x = -5.510×10^{-1}

Note that even a polynomial of same grade deviating just a bit (e.g. $7x^3 + 4.5x^2 - 3x - 2 = 0$) may feature one *real* root only.

Look into *Section 5* of the *HP-27S OM* for more about interactive solving of equations.

The Interactive Solver for Expressions Stored in Programs

Instead of operating on an equation as described in previous chapters, your *WP 43S* can also solve an expression ***f*** stored in a program. Then, the procedure is as follows:

1. Write a program for ***f***:
2. Press ADV SOLVE.
3. Enter values for all known variables of ***f***.
4. Let your *WP 43S* compute the unknown variable.
5. Leave the *Solver*.

We will go through this step by step:

1. Write a program for ***f***:
   - It must begin with a global label.

- It must define all variables required for calculating ***f***.
- It shall be as efficient as possible since it is going to be executed many times.

For interactive solving, proceeding as follows is recommended for this program: From its 2nd step on, menu variables shall be declared using MVAR instructions (cf. p. 222) covering all variables of ***f***. The subsequent body of the routine shall evaluate ***f*** recalling these variables. For a *Solver* routine, the original expression shall be rewritten in a way that ***f*** = 0 is fulfilled.

Example:

Let's return to the equation we dealt with in the last two chapters:

$$height = v_0 \cdot time - g_\oplus / 2 \cdot time^2$$

This is easily rewritten:

$$v_0 \cdot time - g_\oplus / 2 \cdot time^2 - height = 0$$

So the required program might look like this:

```
LBL 'FreeF'
    MVAR 'height'
    MVAR 'v0'
    MVAR 'time'
    # g⊕              take this out of CONST.
    -2
    /
    RCL× 'time'
    RCL+ 'v0'
    RCL× 'time'
    RCL- 'height'     now we have got f.
    RTN
```

2. Press ADV. You will see:

Choose SOLVE. You will get (as expected from p. 210):

Press **PROG** and pick the proper program for ***f*** (here **FreeF**). You will get the corresponding menu of variables, i.e. here:

3. Enter values for all known variables of ***f*** and (optionally) one or two guesses for the unknown.

   In our example, we may just take the values we know from above:

   –50 height

   20 v_0

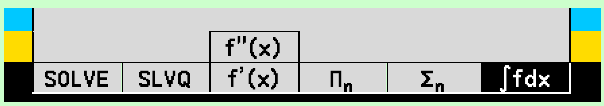
   5 time 10 time

4. Let your *WP 43S* compute the unknown variable.

   Press **time** once more but without a heading numeric entry. Your *WP 43S* will return **time = 5.8** as you have expected (cf. p. 241).

5. Leave the *Solver*.

   Pressing **EXIT** will return to the top *view* of ADV.



Using the Solver in a Program

For using the *Solver* in a programs, it has to be told what you did tell it in the examples of previous chapters. Thus, when you press **ADV** in *PEM*, you will see a slightly different menu than the one you have seen above:

PGMSLV		f"(x)			PGMINT
SOLVE	SLVQ	f'(x)	Π_n	Σ_n	∫fdx

PGMSLV is for specifying the program calculating ***f***. It must be found in your program before SOLVE is called.

Furthermore, define the necessary variables in advance and load them with the known values using STO. Eventually, the unknown variable must be specified calling SOLVE.

Example:

Let's return to the equation we dealt with in the last chapters. So the required program for ***f*** might look like this (like the previous program but without the MVAR steps):

`LBL 'FreeFp'`	
`# g⊕`	take this out of CONST.
`-2`	
`/`	
`RCL× 'time'`	
`RCL+ 'v0'`	
`RCL× 'time'`	
`RCL- 'height'`	now we have got ***f***.
`RTN`	

The program one level above could contain a section looking like this:

`...`	
`PGMSLV 'FreeFp'`	specify the function to be solved.
`SOLVE 'time'`	solve for time.
`VIEW 'time'`	display the solution.
`...`	

Before starting this program (let's call it **C**), fill the variables of the equation to be solved, e.g. with the start values known from above:

−50 STO height
20 STO v_0

Option: Fill the unknown with a 1st guess, e.g. with **5** as we specified above (a 2nd guess will be taken from **X**):

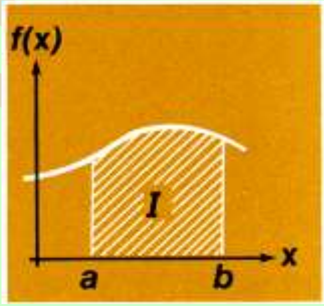
5 STO time

Call **C** via XEQ C and you will see time = 5.8 (as expected from p. 241).

Eventually turn to *Part 3*, *Section 12* of the *HP-42S OM*. Refer to the *HP-34C OHPG* (*Section 8* and *App. A*) or the *HP-15C OH* (*Section 13* and *App. D*) for more information about automatic root finding and some caveats.

Numeric Integration of Equations

The command ∫ lets your *WP 43S* compute definite integrals numerically.



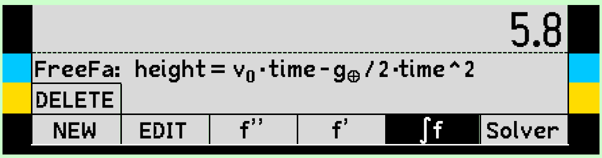
Example:

Let's compute the *Bessel function* of 1st kind and order 0. This function can be written as

$$J_0(x) = \frac{1}{\pi}\int_0^{\pi} cos(x \sin t)dt$$

Solution:

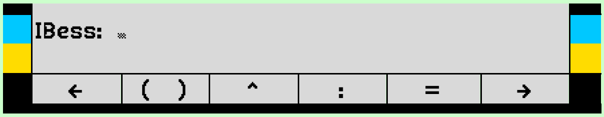
This is calculated in *radians*, thus enter MODE RAD and press EQN :



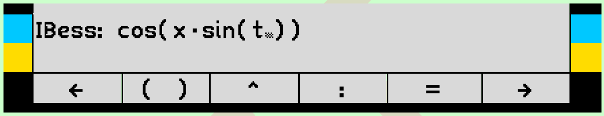
This function is not in the equation list yet.[172] So, press NEW and start entering the integrand:

[172] Actually, a function $J_y(x)$ is implemented in your *WP 43S* directly returning values of the Bessel function of 1st kind and order ***y***. Feel free to compare the results.

I B ▼ E S S :



Continue with C O S ( ) ← X x S I N ( ) ← T



Close and store this function by pressing ENTER↑. The *menu* will return to the previous one.

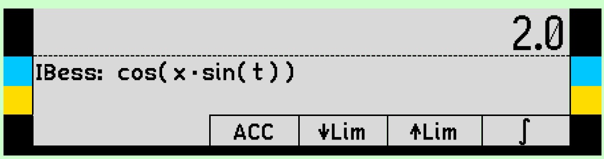
Then press ∫f . Your *WP 43S* will check the current equation (cf. pp. 238ff) for syntax errors (missing operators, misspelled functions, illegal variable names, etc.).[173] It will then return a *menu* of all applicable variables:

You can enter values for any variables (i.e. integration constants) you already know by pressing the respective *softkeys* now, e.g.

2 x

(For recalling such an integration constant, just press RCL VAR before the respective *softkey*.)

Then select the variable of integration by simply pressing t here (there must not be any numeric input heading t). The *menu* will change:



[173] You will have noticed already that **IBess** is not an equation but just one side of it. To keep the system lean, such functions are listed under EQN nevertheless, but cannot be evaluated by the Solver, of course.

Even your *WP 43S* cannot compute an integral exactly, it approximates its value numerically. The accuracy of this approximation depends on the accuracy of the integrand's function itself as calculated by your program. This is affected by round-off error in the calculator and also by the accuracies of the integration constants specified.

ACC is a *real number* that defines the relative error of the integration. With **ACC** = 0.001, for example, you can be sure that

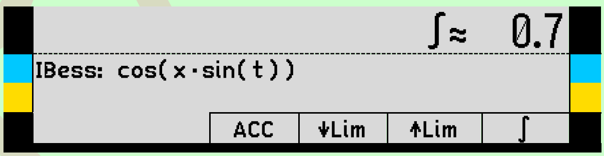
$$\left|\frac{v_T - v_C}{v_C}\right| \leq 0.001$$

(with $\boldsymbol{v_T}$ being the true value and $\boldsymbol{v_C}$ the computed value of the integrand) at any point between **↓Lim** and **↑Lim**.

We want to see the result accurate to three decimals. Thus we enter

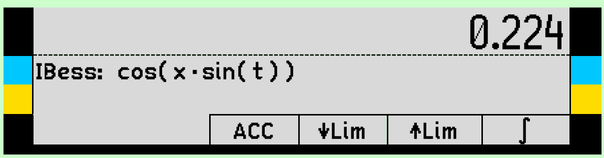
.001 ACC	for the accuracy of computation,
0 ↓Lim	for the lower integration limit,
π ↑Lim	for the upper integration limit,

and start integrating by pressing ∫ . Your *WP 43S* will return:



Do not forget to divide this result by π to get the correct value for $J_0(2)$:

DISP FIX 3 π / [174]



Enter other values for ***x*** and integrate again to get $J_0(x)$ at other locations.

---

[174] Note that ∫≈ vanishes with π like every *temporary information* disappears with the next keystroke.

We could have included that division by π in our function **IBess**. We did not, however, since **IBess** is evaluated many times during the integration process; thus the fewer steps the integrand contains the faster the result can be returned.

Interactively Integrating Expressions Stored in Programs

Instead of operating on an '<u>equation</u>' as described in previous chapter, your *WP 43S* can also integrate an <u>expression **f** stored in a program</u>. Then, the procedure is as follows:

1. Write a program for ***f***.
2. Press ADV ∫fdx.
3. Enter values for all known variables (integration constants) of ***f***, for ACC, and for the integration limits. Select the variable of integration.
4. Let your *WP 43S* compute the definite integral specified.[175]

We will go through this step by step:

1. Write a program for ***f***:
   - It shall begin with a global label.
   - It shall define all variables required for calculating ***f***.
   - It shall be as efficient as possible since it is going to be executed many times.

   It is recommended proceeding as follows: From the 2nd step of this program on, menu variables shall be declared with MVAR instructions (cf. p. 222) covering all variables of ***f***. The subsequent body of the routine shall evaluate ***f*** recalling these variables.

   Example:

   Let's return to the integrand we dealt with in the last chapter. Then the required program for ***f*** might look like this:

```
LBL 'IBessI'
   MVAR 'x'
   MVAR 't'
   RCL 't'
   sin
   RCL× 'x'
   cos
   RTN
```

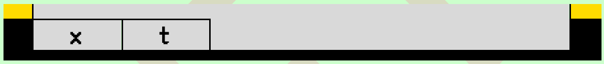
now we have got ***f***.

[175] Note this follows closely the procedure as described for the *Solver* above.

2. Press ADV. You will see:

Choose ∫fdx. You will get (as expected from p. 210):

Press PROG and pick the proper program for ***f*** (here IBessI). You will get the corresponding menu of variables, i.e. here:



3. Enter values for all known variables (integration constants) of ***f*** and select the variable of integration.

   In our example, we may just take the values we know from above: **2** x t. So **t** will be the variable of integration. The *menu* will change now:

   We enter (like in previous chapter) **.001** ACC **0** ↓Lim π ↑Lim .

4. Let your *WP 43S* compute the definite integral specified.

   Press ∫ to integrate with all the parameters as chosen, and your *WP 43S* will return ∫≈ 0.704 as you might have expected (cf. previous chapter). Divide by π to get the value for $J_0(2)$ as above.

Using the Integrator in a Program

For using the Integrator in programs, it has to be told what you did tell it in the examples of the two previous chapters. Thus, when you press

ADV in *PEM*, you will see a slightly different menu than the one you have seen above:

PGMSLV		f"(x)			
SOLVE	SLVQ	f'(x)	Π_n	Σ_n	∫fdx

You shall define the necessary variables in advance and load them with the known values using STO. Then call the *menu* ∫fdx. PGMINT is for specifying the program calculating ***f***. It must be found in your program before the integration itself is called. And the integration limits as well as the requested accuracy shall be stored as well before integrating:

PGMINT		STO AC	STO ↓L	STO ↑L	∫

Eventually, the variable of integration must be specified calling ∫.

Example:

Let's return to the integrand we dealt with in the last two chapters. So the required program for ***f*** might look like this:

LBL 'IBessP'	
RCL 't'	
sin	
RCL× 'x'	
cos	now we have got ***f***.
RTN	

The program one level above could contain a section looking like this:

...	
PGMINT 'IBessP'	specifying the function to be integrated.
0	
STO '↓Lim'	
π	
STO '↑Lim'	
0.001	
STO 'ACC'	
∫fd 't'	integrate over time.
VIEW ST.X	display the solution.
...	

Before starting this program (let's call it **InP**), fill the variables staying constant under integration, e.g. with the start values known from above:

2 STO x .

Call **InP** via XEQ and you will see ∫≈ 0.704 as you may have expected (cf. previous chapter). Divide by π to get $J_0(2)$ as above.

Eventually turn to Part 3, Section 13 of the *HP-42S OM*. Refer to the *HP-34C OHPG* (*Section 9* and *App. B*) or the *HP-15C OH* (*Section 14* and *App. E*) for more information about automatic integration and some caveats.

Differentiating Equations

There are two commands provided returning the values of the first two derivatives of the function $f(x)$ at position x. This function $f(x)$ can be specified in an equation.

f'(x) returns the 1st derivative. For computing it, ...

1. f'(x) will first look for a user routine labeled '**δx**' (or '**δX**', '**Δx**', or '**ΔX**', in this order), returning a fixed step size ***dx*** in **X**. If that routine is not defined, ***dx*** = 0.1 is set for default.
2. Then, f'(x) fills all *stack registers* with x and calls $f(x)$. It will evaluate $f(x)$ at ten points equally spaced in the interval $x \pm 5\ dx$ (if you expect any irregularities within this interval, change ***dx*** to exclude them).
3. On return, the 1st derivative will be in *stack register* **X**, while **Y**, **Z**, and **T** will be clear and the position x will be in **L**.

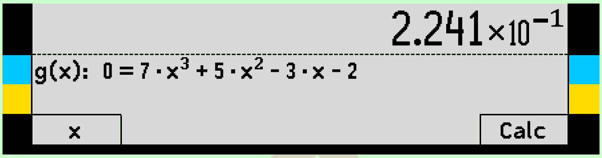
Example (with SCI 3 set):

Take the equation $g(x) = 7x^3 + 5x^2 - 3x - 2$ again (used on pp. 241f for solving). Instead of checking two function values left and right of the root you could check the slope $g'(x)$ at the root just once.

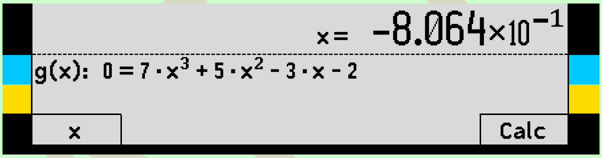
Solution:

You have got ***g***(*x*) in EQN already. For each of the three roots found, calculate the root first, then the 1st derivative of ***g***(*x*) at that point:

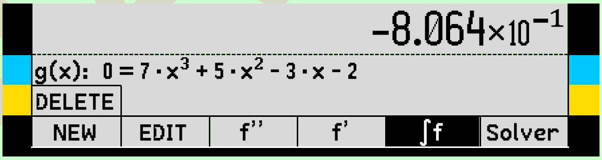
1. Press EQN, make ***g***(*x*) the *current equation*, and press Solver. You will see then:



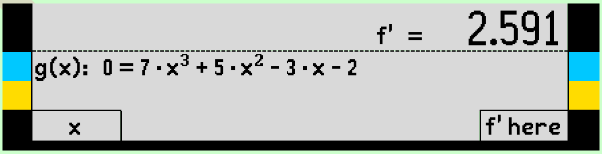
2. Find the 1st (leftmost) root as shown above:
   −2 x **−1** x x



3. Pressing EXIT returns to the top *view* of EQN:



4. Press f' :



   Note that f' returned the value of the 1st derivative at this very location immediately since ***g***(*x*) features only one variable; else f' would have needed your input via the *softkeys* displayed and pressing f' here thereafter.

   So the slope of ***g***(*x*) at ***x* = −0.8064** is **2.591**. Get the slopes at the two other root positions the same way:

5. EXIT returns to the top *view* of EQN as in step 3.

6. Solver
Find the 2nd root of $\boldsymbol{g}(x)$: **−1** x **0** x x

x = -5.510×10^{-1}

EXIT returns to the top *view* of EQN as in step 3.
f'

f' = -2.134

EXIT returns to the top *view* of EQN as in step 3.

7. Solver
Find the third (rightmost) root of $\boldsymbol{g}(x)$: **0** x **1** x x

x = 6.431×10^{-1}

EXIT returns to the top *view* of EQN as in step 3.
f'

f' = 1.211×10^{1}

So the slope of $\boldsymbol{g}(x)$ at $\boldsymbol{x} = \mathbf{-0.8064}$ is **2.591**, at $\boldsymbol{x} = \mathbf{-0.5510}$ it is **−2.134**, and at $\boldsymbol{x} = \mathbf{0.6431}$ it is **12.11**; the sequence of slopes is positive, negative, and positive as expected.

f”(x) works in full analogy, computing the 2nd derivative of the function specified.

Interactively Differentiating Expressions Stored in Programs

Instead of operating on an ‘equation’ as described in previous chapter, your *WP 43S* can also derive an expression $\boldsymbol{f}(x)$ stored in a program. Then, the procedure works as follows:

1. Write a program for $f(x)$. It must begin with a global label. For interactive derivation, proceeding as follows is recommended: From the 2nd step of this program on, menu variables shall be declared with MVAR instructions (cf. p. 222) covering all variables of $f(x)$. The subsequent body of the routine shall evaluate $f(x)$ recalling these variables.
2. Optionally, write another program labeled '**δx**' (see p. 254).
3. Enter values for all known variables (derivation constants) of $f(x)$. Put the location where you want to know he derivative into **X**.
4. Press ADV. You will get:

5. Press f'(x) or f"(x). You will get as well:

6. Press PROG to pick the label of the program containing the function $f(x)$ (or enter its label directly as described on p. 209).
7. Let your *WP 43S* compute the 1st or 2nd derivative at the location specified in x.[176]

Computing Derivatives in a Program

For computing derivatives in programs, proceed as demonstrated in previous chapter. Just remember you should omit the MVAR instructions in your program calculating $f(x)$; instead, define the necessary variables in advance and load them with the known values using STO.

---

[176] Note this follows loosely the procedures as described for the Solver and Integrator above.

When you press ADV in *PEM*, you will see a slightly different menu than the one you have seen above:

Press f'(x) (or f"(x)) and specify the label of your program calculating $\boldsymbol{f(x)}$ – or pick it from the list as explained in steps 5 and 6 above. Your *WP 43S* will compute the requested derivative for you in this program step.

Nesting Advanced Operations

You can nest SLV, ∫, f'(x), f''(x), Σ, and Π in routines to any depth as far as memory allows and your patience and power last.

Example:

Light is observed to be diffracted when passing through small circular holes, an effect most obvious when using laser light. Its intensity is $I(r) = I_0 \times \left(\frac{J_1(2\pi r)}{\pi r}\right)^2$ behind the hole; $J_1(x) = \frac{1}{\pi}\int_0^{\pi} cos[\, t - x\, sin(t)]\, dt$ is the *Bessel function of the 1st kind of order 1* (cf. p. 248). Find the first three roots of the intensity (i.e. the radii where no light will be observed).

Solution:

1. Write a little program for the internal calculation of the integrand $f(t) = cos[\, t - x\, sin(t)]$:

LBL 'J1'	
sin	$sin(t)$
RCL× 00	The entire stack is loaded with the integration variable $\boldsymbol{t}$, so $\boldsymbol{x} = 2\pi r$ (see below) must be recalled from a global *register* for calculating $x\, sin(t)$
-	$t - x\, sin(t)$
cos	$cos[\, t - x\, sin(t)]$
RTN	

.

2. Write a 2nd little program for the internal calculation of the intensity $I(r)$. Note that just the parenthesis of the formula above must be evaluated since I_0 is a constant. And ADV, when called in *PEM*, displays:


```
LBL 'I'
    π
    ×
    2
    ×
    STO 00          stores 2πr for later use, also in integration.
    PGMINT 'J1'     specifies the program of the integrand.[177]
    0
    STO '↓Lim'
    π
    STO '↑Lim'
    0.001
    STO 'ACC'
    ∫fd ST.X        computes π × J1(2πr) .
    RCL 00          the stack just contained the integration results before.
    /               J1(2πr) / (2πr)
    2
    ×               J1(2πr) / (πr)
    RTN
```

3. Enter DISP SCI 3
   MODE RAD
   0.1 ENTER↑ **1** ADV SOLVE

4. Enter I. You will get 6.098×10^{-1} after some time.[178]

---

[177] You shall press the rightmost softkey to get the *menu* for accessing PGMINT etc.

[178] I.e. after some 25*s* using the *WP 43S* emulator (cf. *ReM*, *App. I*) on a PC running *Windows 10*. It will take xxx minutes on your calculator. Nesting advanced operations may require very many of calculations to be performed! We recommend connecting

5. Enter **1** ENTER↑ **1.5** SOLVE [] and you will get 1.117 .

6. Enter **1.5** ENTER↑ **2** SOLVE [] and you will get 1.619 .

You will find further instructions and examples in *HP-42S RPN Scientific Programming Examples and Techniques.* Despite the title of this manual, it also contains significant material about the *Solver*, integration, matrices, and statistics.

your *WP 43S* to an *USB* outlet for external power supply when dealing with such applications. – Note the Solver was not started at **0** since that would cause an error when dividing by $2\pi r$.

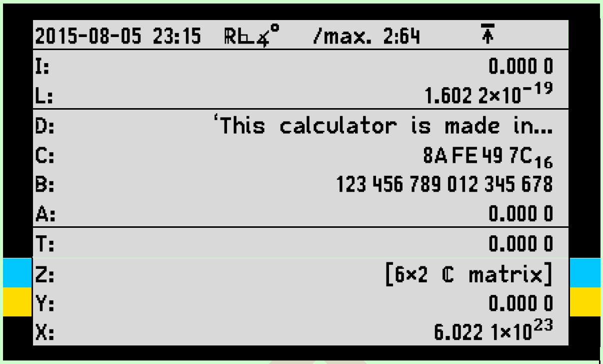
SECTION 5: TWO BROWSERS, TWO APPLICATIONS, AND TWO SPECIAL MENUS

There are two *browsers* featured for quick and easy checking memory, *registers*, and *flags* (RBR and STATUS, see below). And there are two very useful applications: a TIMER (or stopwatch, see pp. 264f) and "*time value of money*" (TVM, see pp. 266ff). Furthermore, two special *menus* will ease your path in special areas of application and particular regions of this planet (see pp. 270ff).

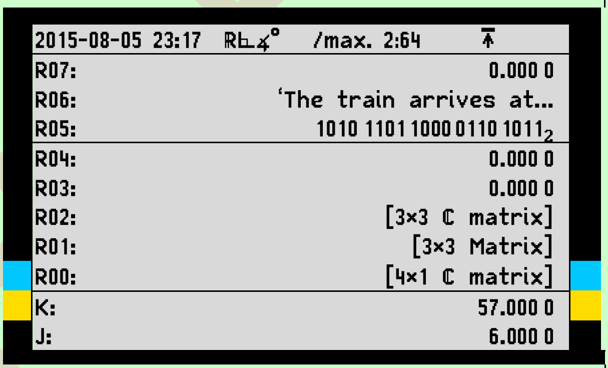
The Browsers RBR and STATUS

These two *browsers* may be called in all modes except *alpha input*. Some special keys and special rules apply within these *browsers* as explained on the two pages following. EXIT works as in *menus*, however, leaving the respective *browser* now; and *browser* (like *menu*) calls cannot be programmed.

Keys to press	Contents and special remarks
f RBR	Browses all currently allocated *registers* showing their contents. RBR operates in *TAM* (cf. pp. 56ff). The first screen you see covers *registers* **X** through **I** (their contents will deviate on your screen – note numeric contents are shown explicitly in the display format currently set as long as they are individual numbers while strings may be abbreviated and matrices will be. Fractions are displayed with their decimal value. Within the range of lettered *registers*, every fourth *register* is displayed overlined to guide the eye:



▲ goes up the *stack*, continuing with the remaining lettered *registers*, then with **R00**, **R01**, etc. as shown below. For **R00** … **R99**, every fifth *register* is displayed overlined to guide the eye. After **R99**, **X** will be shown again:



▼ browses the *registers* going down from **R99** (if starting with the screen on previous page) to **R00**; then continues with **K**, **J**, down to **X**. After **X**, **R99** will be shown again.

. turns to *local registers* if allocated, starting with **R .00**. Then, ▲ and ▼ browse *local registers* up and down until another . returns to the first screen of RBR as shown on p. 261. Else (i.e. if no *local registers* are allocated) . directly returns to this screen.

<table>
<tr><td>0 0 …
9 9</td><td>browses immediately to the corresponding (global or local) register. If no such local register is allocated, it returns to the first global register.</td></tr>
<tr><td>R/S</td><td>toggles display to show the <u>r</u>egister contents or the <u>s</u>pace allocated for them.</td></tr>
<tr><td>RCL</td><td>in run mode, recalls the register displayed in the lowest row and leaves RBR; in PEM, enters a corresponding step RCL … and leaves RBR.</td></tr>
<tr><td>EXIT</td><td>leaves RBR.</td></tr>
<tr><td>STATUS

or

g FLAGS
STATUS</td><td>Displays the amount of free memory available and the user accessible flags set (inspired by STATUS on HP-16C and WP 34S). Local user flags will only be displayed if local registers are allocated at all. Some global settings and system flags set are shown in the bottom rows (covering only what is not shown in the status bar):

</td></tr>
<tr><td>▲ and ▼</td><td>toggle between views if more flags are set.</td></tr>
<tr><td>EXIT</td><td>leaves STATUS.</td></tr>
</table>

☞ <u>No other keys</u> will work within RBR and STATUS. And both browsers are not programmable.

The Timer Application

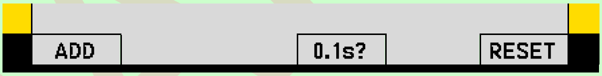
Your *WP 43S* provides a timer following the one of the *HP-55*.[179] Start it by pressing TIMER. Then the top numeric row will be replaced by

`0:00:00.0 [00]`

or (depending on the radix mark setting)

`0:00:00,0 [00]`

unless the timer was running before already (then the accumulated run time will be indicated instead of zero here). In either case the *menu section* will change to this:



Within TIMER, only the following keys will work:

Key	Remarks
R/S	starts or stops the timer without changing its value.
RESET	resets the timer to zero without changing its status (running or stopped). It deletes the total time if applicable. – Note this is not the global RESET command.
0.1s?	toggles displaying tenths of *seconds* (default is 'display').
ADD	adds the present timer value to the statistics *registers*. This allows for computing e.g. the *arithmetic mean* and *standard deviation* of lap times after leaving TIMER.

[179] This application works exactly as in the *WP 34S* but the display differs. With respect to the *HP-55*, there are two deviations:

1. Your *WP 43S* will not take the content of **X** at the time calling TIMER as start time of the timer; start times are supported by RCL within TIMER here instead.
2. Your *WP 43S* will display tenths instead of hundredth of *seconds*. Reaction times of the hardware do not allow for more precision anyway.

Key	Remarks
n n	sets the *current register address* (*CRA*, *startup default* is 0). The *CRA* will be displayed between rectangular brackets as shown here:[180] 27:31:55.6 [01]
ENTER↑	stores the present timer value in the *current register* at execution time without changing the timer status or value. Then increments the *CRA* by one.
RCL ***nn***	recalls ***rnn*** without changing the status of the timer. The value recalled may be used e.g. as start time for further incrementing.
▲ or ▼	increments or decrements the *CRA* by one, respectively. ▼ allows for overwriting the last value stored.
.	combines ENTER↑ + ⇐ in one keystroke, but the total time since the last explicit press of ⇐ or RESET is shown and updated like: 21:04:15ᵀ 0:02:29 [06] or 10:02:31.7ᵀ 0:00:49.6 [11] . . allows for recording lap times, for example. Note the total time is volatile – it will disappear without a trace when ⇐ or RESET is pressed alone.
+	Combines all the functionality of ADD, ENTER↑, and ⇐ in one keystroke. This allows for recording lap times and total time for later offline analysis.
EXIT	leaves the application. The time indicated in the top numeric row will vanish from the screen. Unless already stopped, however, the timer continues incrementing in the background (indicated by ◷ in the *status bar*) until …

[180] Think about specifying the *CRA* so there will be sufficient unused *registers* following. Attempts to specify a *CRA* <0 or >99 will be blocked.

On the *HP-55*, input of a single digit sufficed for storing, since only 10 *registers* were featured for this purpose there. Furthermore, there was no automatic address increment.

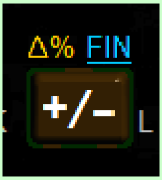
Key	Remarks
	a) stopped explicitly by **R/S** within TIMER or ... b) your *WP 43S* is turned off by you. Note it will <u>not turn off automatically</u> with the timer running. A reliable power supply is recommended in such cases.

☞ For subtracting split times you have to leave this application.

☞ TIMER is not programmable.

The Time Value of Money (TVM)

TVM is a well proven financial application (thus found in <u>FIN</u>) computing e.g. the future value (***FV***) of



1. a repeated investment or
2. a regular down-payment for a credit

based on its present value (***PV***), its interest rate per annum (***i%/a***), the required payment per period (***PMT***), number of periods per annum (***per/a***), and the total number of payment periods (***n***$_{PER}$). This kind of financial problems will often occur also to technical people, so we included TVM on your *WP 43S*.[181]

For your information, the general formula for such problems reads

$$FV = PV \times (1+i)^{n_{PER}} + (1 + i \times p)\frac{PMT}{i} \times \left[1 - (1+i)^{n_{PER}}\right]$$

with the deduced parameter $i = \frac{I\%/period}{100} = \frac{I\%/year}{100} / \frac{periods}{year}$

---

[181] *TVM* was launched with *HP*'s very first financial '*problem solver*', the *HP-80* of 1972, and was implemented on each and every *HP* '*business calculator*' thereafter. An early advertising sheet promised 'you can improve and simplify your time-and-money management' applying TVM quickly. 'Random-entry financial keys let you key in problems in <u>any</u> order. And you can change <u>any</u> number at <u>any</u> time.' All this was true for certain (remember it was a time before spreadsheet software became available) and may even hold nowadays to some extent since hardly any modern financial tool solving such problems is more compact than a pocket calculator featuring *TVM*.

and the binary switch value p: If payments occur at the

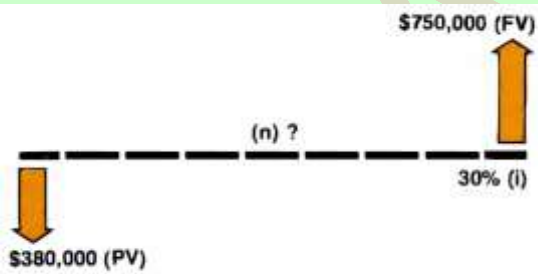
- end of each period then $p = 0$ (choose **End** in TVM).
- beginning of each period then $p = 1$ (choose **Begin** in TVM).

***TVM* uses the convention that cash outlays are input as negative, and cash incomes are input as positive.**

The present value ***PV*** always occurs at the beginning of the 1st period. It can also be an initial cash flow or a discounted value of a series of future cash flows.

The future value ***FV*** is always meant to occur at the end of the n_{PER}th period. It can also be a final cash flow or a compounded value of a series of cash flows.

Example for calculating the number of periods (from the *HP-27 OH*, like all following examples in this chapter; enjoy the amounts and interest rates of a time long ago):



A potential development site currently appraised at $380 000 appreciates at 30% per year. If this rate continues, how many years will it be before this land is worth $750 000?

Solution:

DISP FIX 2 — This will suffice. Then all you have to do is keying in the known parameters and boundary conditions:

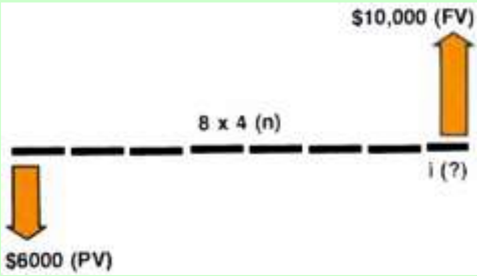
FIN TVM Begin

Begin					End
n_{PER}	i%/a	per/a	PV	PMT	FV

380000	PV	380 000.00	present value,
30	i%/a	30.00	% interest rate per year,
0	PMT	0.00	no payments,
1	per/a	1.00	default.

750000 FV	750 000.00	Now, how long does it take to reach this future value?
n_{PER}	2.59	years.

Example for finding the necessary interest rate for compounded amounts:

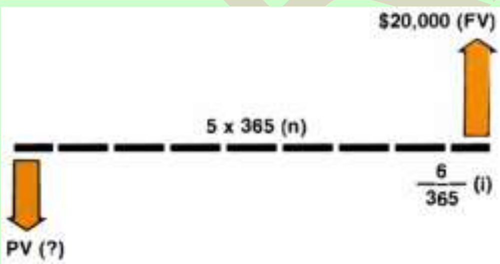


What annual interest rate must be obtained to amass $10 000 in 8 years on an investment of $6 000, with quarterly compounding? (Continue keeping the settings of previous example.)

Solution:

6000 PV	6 000.00	present value,
4 per/a	4.00	quarters,
10000 FV	10 000.00	future value,
8 ENTER↑ 4 × n_{PER}	32.00	periods. Now, we need...
i%/a	6.44	% interest rate per year to achieve this.

Example for finding the present value of a compounded amount:

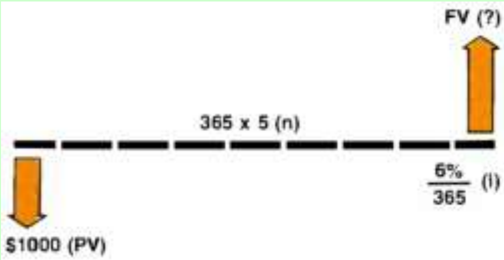


In 5 years when your son starts college, you will need $20 000. You deposit a lump sum in a certificate account that earns 6% compounded daily. How much do you need to deposit today to reach that goal? (Dream on with the settings of previous example.)

Solution:

20000 FV	20 000.00	future value,
6 i%/a	6.00	% interest rate per year,
365 per/a	365.00	days per year,
5 ENTER↑ 365 × n_{PER}	1 825.00	periods. Now, we need...
PV	14 816.73	to be deposited.

Example for finding the future value of a compounded amount:



The local trading post manager opened up a savings operation 5 years ago, offering 6% compounded daily. Gold miner Yellowstone Sam deposited $1000 at that time, and now wants to know his present balance and the total accrued interest after all this time. (Continue dreaming ...)

Solution:

1000 PV	1 000.00	original deposit,
6 i%/a	6.00	% interest rate per year,
365 per/a	365.00	days per year,
5 ENTER↑ **365** × n_{PER}	1 825.00	periods. Now, Sam has...
FV	1 349.83	present balance meaning...
RCL VAR PV −	349.83	accrued interest.

Nominal interest rate converted to effective rate:

Example for finding the effective annual interest rate:

What is the effective annual rate of interest if the annual nominal rate of 12% is compounded quarterly? (Continue keeping the settings of previous example.)

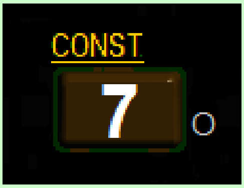
Solution:

100 PV	100.00	base value,
12 i%/a	12.00	% nominal rate per year,
4 per/a	4.00	quarters per year,
4 n_{PER}	4.00	compound periods;
FV	112.55	
RCL VAR PV −	12.55	% effective interest rate.

Turn to *App. 3* for more applications of TVM (annuities, savings, etc.), starting on p. 310.

Constants

Your *WP 43S* contains a *catalog* of 80 physical, astronomical, and mathematical constants, sorted alphabetically in CONST. Press CONST and the *menu section* will change to:



Besides by browsing with ▲ and ▼, you can access the contents of CONST most easily using the alphabetical access method demonstrated in the *ReM*, *Section 2*.

Names of astronomical and mathematical constants are printed on colored background in the table starting below. The unit of each physical and astronomical constant is listed here as well. Find the numeric values of the constants and their uncertainties in the *ReM*, *Section 2*.

Name	Unit [182]	Remarks
a	d	*Gregorian* year
a_0	m	*Bohr* radius
a_{Moon}	m	Semi-major axis of the Moon's orbit around the earth.
$a_\oplus$	m	Semi-major axis of the Earth's orbit around the sun. Within its uncertainty, $a_\oplus$ equals 1 *AU* (*astronomic unit*).
c	$^m/_s$	Speed of light in vacuum
c_1	$m^2\,W$	First radiation constant
c_2	m K	Second radiation constant
e	C	Elementary charge
e_E		*Euler's e*

[182] Find all unit symbols used here explained in the chapter about unit conversions in the *ReM*, *Section 2*.

Name	Unit [182]	Remarks
F	C/mol	*Faraday* constant
F_α F_δ		*Feigenbaum's* α and δ
G	m^3/kg s^2	*Newtonian* constant of gravitation; also known as γ from other authors. See also $GM_\oplus$ below.
G_0	1/Ω	Conductance quantum
G_C		*Catalan's* constant
g_e		*Landé's* electron g-factor
$GM_\oplus$	m^3/s^2	*Newtonian* constant of gravitation times the Earth's mass with its atmosphere included (according to the Earth model *WGS84* – see the *ReM* for more information)
$g_\oplus$	m/s^2	Standard earth acceleration
h	J s	*Planck* constant
ħ	J s	So-called '*Dirac* constant', actually only h over 2π
k	J/K	*Boltzmann* constant
K_J	Hz/V	*Josephson* constant
l_P	m	*Planck* length
m_e	kg	Electron mass
M_{Moon}	kg	Mass of the Earth's Moon
m_n	kg	Neutron mass
m_p	kg	Proton mass
M_P	kg	*Planck* mass
m_p/m_e		Proton to electron mass ratio
m_u	kg	Atomic mass constant
m_uc^2	J	Energy equivalent of atomic mass constant

<table>
<tr><th>Name</th><th>Unit [182]</th><th colspan="2">Remarks</th></tr>
<tr><td>m_μ</td><td>kg</td><td colspan="2">Muon mass</td></tr>
<tr><td>$M_\odot$</td><td>kg</td><td colspan="2">Mass of the Sun</td></tr>
<tr><td>$M_\oplus$</td><td>kg</td><td colspan="2">Mass of the Earth. See also **$GM_\oplus$** above.</td></tr>
<tr><td>N_A</td><td>$^1/_{mol}$</td><td colspan="2">*Avogadro's* number</td></tr>
<tr><td>**NaN**</td><td></td><td colspan="2">"Not a Number", i.e. e.g. $^0/_0$ or $\pm\infty \times 0$ or $ln(x)$ for $x < 0$ or *tan*(90°) unless in *complex* domain.
NaN covers poles as well as regions where a function result is not defined at all. Note that infinities, on the other hand, are considered numeric in your *WP 43S* (see the end of this table). Non-numeric results will lead to an error message thrown – unless SPCRES is set. **NaN** allows that functions written by you can return it.</td></tr>
<tr><td>p_0</td><td>Pa</td><td colspan="2">Standard atmospheric pressure</td></tr>
<tr><td>R</td><td>$^J/_{mol\ K}$</td><td colspan="2">Molar gas constant</td></tr>
<tr><td>r_e</td><td>m</td><td colspan="2">Classical electron radius</td></tr>
<tr><td>R_K</td><td>Ω</td><td colspan="2">*Von Klitzing* constant</td></tr>
<tr><td>R_{Moon}</td><td>m</td><td colspan="2">Mean radius of the Moon</td></tr>
<tr><td>R_∞</td><td>$^1/_m$</td><td colspan="2">*Rydberg* constant</td></tr>
<tr><td>$R_\odot$</td><td>m</td><td colspan="2">Mean radius of the sun</td></tr>
<tr><td>$R_\oplus$</td><td>m</td><td colspan="2">Mean radius of the Earth</td></tr>
<tr><td>Sa</td><td>m</td><td>Semi-major axis</td><td rowspan="5">… according to *WGS84* (see the *ReM*)</td></tr>
<tr><td>Sb</td><td>m</td><td>Semi-minor axis</td></tr>
<tr><td>Se^2</td><td></td><td>First eccentricity squared</td></tr>
<tr><td>Se'^2</td><td></td><td>Second eccentricity squared</td></tr>
<tr><td>Sf^{-1}</td><td></td><td>Flattening parameter</td></tr>
<tr><td>T_0</td><td>K</td><td colspan="2">= 0°C, standard temperature</td></tr>
</table>

Name	Unit [182]	Remarks
t_p	s	*Planck* time
T_p	K	*Planck* temperature
V_m	m^3/mol	Molar volume of an ideal gas at standard conditions ≈ 22.4 l/mol
Z_0	Ω	Characteristic impedance of vacuum
α		Fine-structure constant
γ	$m^3/kg\ s^2$	*Newtonian* constant of gravitation; also known as **G** from other authors. See also $GM_\oplus$ above.
γ_{EM}		*Euler-Mascheroni* constant
γ_p	Hz/T	Proton gyromagnetic ratio
$\Delta\nu_{Cs}$	Hz	Hyperfine transition frequency of ^{133}Cs
ε_0	F/m	Electric constant or vacuum permittivity
λ_C	m	*Compton* wavelengths of the electron, neutron, and proton
λ_{Cn}		
λ_{Cp}		
μ_0	H/m	Magnetic constant or vacuum permeability
μ_B	J/T	*Bohr* magneton
μ_e		Electron magnetic moment
μ_e/μ_B		Ratio of electron magnetic moment to *Bohr's* magneton
μ_n	J/T	Neutron and proton magnetic moment
μ_p		
μ_u		Nuclear magneton
μ_μ		Muon magnetic moment
σ_B	W/m^2K^4	*Stefan-Boltzmann* constant

Name	Unit [182]	Remarks
Φ		Golden ratio
Φ_0	Wb	Magnetic flux quantum
ω	rad/s	Angular velocity of the Earth according to *WGS84* (see the *ReM*)
$-\infty$		May the Lord of Mathematics forgive us calling these two constants! Both are counted as special numeric values in your *WP 43S*, however.
∞		

Employ the constants stored here for further useful equivalences, e.g.:

- express *joules* in *electron-volts* ($1\ \mathrm{J} = 1\ \mathrm{A}\,s\,\mathrm{V} = 1\ ^{\mathrm{eV}}/_{e} \approx 6.24 \times 10^{18}\ \mathrm{eV} = 6.24 \times 10^{9}\ \mathrm{GeV}$),
- calculate the wavelength from the frequency of electromagnetic radiation via $\lambda = {}^{c}/_{\nu}$ (so 1000 THz correspond to ca. 300 nm),
- determine the energy of electromagnetic radiation from its frequency via $E = h\nu$ (so $1\ \mathrm{THz} \times h = 6.63 \times 10^{-22}\ \mathrm{J} = 4.14 \times 10^{-3}\ \mathrm{eV}$). Thus, 1 eV corresponds to 241.8 THz (or a wavelength of 1.24 µm).

Another **example:**

If you want to see the energy equivalent (in *electron-volts*) of one of the small masses given in kg above, multiply its mass by

$$^{c^2}/_{e} \approx 5.610 \times 10^{35}\ ^{\mathrm{m}^2}/_{\mathrm{A\,s}^2}$$

and you are done: m_e corresponds to 511.0 keV, m_p to 938.3 MeV, etc.

One more final **example:**

Assume American advanced scientists will succeed in producing a tiny bit of anti-matter in one of their high-tech laboratories one day – e.g. 0.1 µg of anti-hydrogen, carefully stored isolated in ultra-high vacuum. Although in future, most probably American power transmission lines will still look like they do today since this is a well-tried American (first) standard.

Thus, under slightly extreme weather conditions,[183] an accidental blackout may easily happen for some days – the electric vacuum pumps will stop working, and a subsequent vacuum breakdown will let atmospheric gas leak into the shiny vacuum vessel where it will interact with the precious anti-matter and annihilate immediately. How much energy is going to be released then?

Solution:

You only need the same tiny amount of (usual) matter, so 0.2 μg will annihilate in total within the vessel. 1 $\mu g = 10^{-6}$ g $= 10^{-9}$ kg. Thus enter:

DISP ENG 3	0.000
.2 E +/- 9	$.2\times10^{-9}$
CONST c	299.8×10^{6}
x^2	89.88×10^{15}
×	17.98×10^{6}

… resulting in 18 MJ set free. The odds are frightening high this lab will need no cleaning anymore.[184]

On the other hand, 0.1 μg of anti-matter require e.g. $N_A/10^7$ atoms of anti-hydrogen (with N_A being *Avogadro's* number); this means 6×10^{16} atoms or 3×10^{16} molecules of this gas (i.e. 30 000 million millions molecules). Luckily, this amount is far from being produced in any lab for the time being.[185]

[183] Think of a thunderstorm, blizzard or alike, maybe even fostered by anthropogenic climatic change. Though do not be afraid, this is all fake news created by insane minds according to the greatest president of that blessed nation.

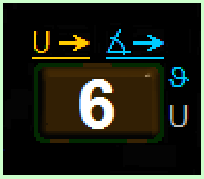
[184] For comparison, 1 kg of *TNT* releases 4.6 MJ. The official definition is some 10% less than this value for historical reasons. Anyway, 18 MJ are equivalent to some 4 kg of *TNT*, enough for a great blast.

[185] And proper *UHV* vessels show very low leak rates as well, so the annihilation energy may be released in little bits over a longer time interval – power supply may be reestablished in time and vacuum pumps operating again. For crucial applications, however, uninterruptible power sources based on batteries and/or generators should be installed locally wherever supplies are threatened by the actual state of public infrastructure being significantly less than great.

And furthermore, ordering antipasti and pasta together in an Italian ristorante is strictly at your own risk. You have been warned!

Unit Conversions

Your *WP 43S* features fourteen angular conversions stored in ∠→ (as shown on p. 126) and 88 unit conversions in U→. The latter *menu* mainly provides means to convert local to common units and vice versa.[186]



Also the constant $\mathbf{T_0}$ may be useful for converting centigrade temperatures to *kelvin*. It is found in CONST and is not repeated in U→ because it is only added or subtracted.

In an attempt to bring some order in that heap of units, U→ is structured like a tree. Press U→ and the *menu section* will change to:

containing the labels of *submenus* for conversions of energy, power, force & pressure, mass, length, area, and volume units. The entire structure of U→ is shown overleaf (with the *menu* rows printed top down instead of bottom up following common reading habits). Some softkeys require more than six characters due to long unit names – then extra high *menu* rows will be displayed:

[186] The *SI* system of coherent units of measurement is agreed on internationally and adopted by almost all countries on this planet for long, as was mentioned above already. Thus, most of the material appearing in U→ will look quirky or obsolete for the overwhelming majority of mankind. Those units die hard, however, in some corners of this world (English is spoken in all of those).

Thus, U→ may also help you when you get caught in a time loop and happen to be thrown back into such an obstinate environment. For symmetry reasons, we think about including some traditional Indian and Chinese units in U→, too.

U→ may also give you a slight idea of the mess we had in the world of measuring before going metric following the French Revolution over 200 years ago. In the *ReM*, you find comprehensive explanations of all conversions provided.

Without *Imperial* and *US*-American units, U→ would contain eighteen entries only.

							Remarks
U→:	E:	P:	year→s	F&p:	m:	x:	submenu headers, units of temperature, time, torque, and ratios – the latter two in an extra-high menu row
	°C→°F	°F→°C	s→year		V:	A:	
	power ratio → dB	dB → power ratio	Nm → lbf×ft	lbf×ft → Nm	field ratio → dB	dB → field ratio	
E:	cal → J	J → cal	Btu → J	J → Btu	Wh → J	J → Wh	units of energy
P:	hp_E → W	W → hp_E	hp_{UK}→W	W→hp_{UK}	hp_M → W	W → hp_M	units of power
F&p:	lbf → N	N → lbf	bar→Pa	Pa→bar	psi→Pa	Pa→psi	units of force and pressure
	in.Hg → Pa	Pa → in.Hg	torr → Pa	Pa → torr	atm→Pa	Pa→atm	
			mmHg → Pa	Pa → mmHg			
m:	lb.→kg	kg→lb.	cwt→kg	kg→cwt	oz → kg	kg → oz	units of mass
	stone → kg	kg → stone	short cwt→kg	kg → sh.cwt	tr.oz → kg	kg → tr.oz	
	ton→kg	kg→ton	short ton → kg	kg → short ton	carat → kg	kg → carat	
x:	au → m	m → au	ly → m	m → ly	pc → m	m → pc	units of length
	mi.→ m	m → mi.	nmi.→m	m→nmi.	ft.→ m	m → ft.	
	in.→ m	m → in.			yd.→ m	m → yd.	
	fathom → m	m → fathom	point → m	m → point	survey $foot_{US}$ → m	m → survey $foot_{US}$	
A:	acre → m^2	m^2 → acre	ha → m^2	m^2 → ha	$acre_{US}$ → m^2	m^2 → $acre_{US}$	units of area
V:	gl_{UK}→m^3	m^3→gl_{UK}	qt.→m^3	m^3→qt.	gl_{US}→m^3	m^3→gl_{US}	units of volume
	$floz_{UK}$ → m^3	m^3 → $floz_{UK}$	barrel → m^3	m^3 → barrel	$floz_{US}$ → m^3	m^3 → $floz_{US}$	

Find out more about the various units mentioned in these conversions in *Section 2* of the *ReM*.

You may combine conversions as you like (DISP ENG 2 will do for all examples in this chapter):

Example 1:

For filling your tires with a maximum pressure of 30 psi the following will help you at gas stations in Europe and beyond:

30 U→ F&p: psi→Pa	returns	207.×10³	Pa.
Pa→bar		2.07	bar.

Now you can set the filler and will not blow your tires.

Example 2:

Your friend tells you she has got 10 *cubic feet* of debris on her veranda after flooding (yes, the dams in the Mississippi delta turn out being of less use than once thought). What does this mean in real units?

1 U→ x: ft.→m	returns	305.×10⁻³	
3 y^x		28.3×10⁻³	
10 ×		283.×10⁻³	m^3.

OK, some work – but manageable.

Example 3:

A network switch is specified for 3 320 Btu/h. What?!?

3320 U→ E: Btu→J	returns	3.50×10⁶	J/h.

Since $1J = c_c Wh \Leftrightarrow 1^J/_h = c_c W$ applies, you can use

J→Wh	for converting and get	973.	W.

This is almost 1 kW. Now you know what will be going on there.

Example 4:

In *Section 2*, there was an example ending with a box featuring a volume of 19 11/16 *cubic inches*. So, what does this volume mean in real units instead? And how much water can such a box contain in areas where people are condemned to deal with *Imperial* units nowadays still?

1 U→ x: ▲ in.→m	returns	25.4×10^{-3}
3 y^x		16.4×10^{-6}
19 . **11** . **16** ×		$323.\times10^{-6}$ m^3.

Since $1\ m^3 = 1000\ liter$, this volume is almost $^1/_3\ liter$.

And to help those enduring life on the *British Imperial* islands or ex-territories, you must (!) ask them for their location first. Then choose either U→ V: $m^3 \rightarrow floz_{UK}$ or $m^3 \rightarrow floz_{US}$ and give them the respective result, i.e. **11.4** or **10.9**, for what it is worth.

Example 5:

A celestial object moves with a velocity of 0.1 *parsec per year*. What does this mean in standard units? What is this in relation to the velocity of light? And how does this translate for air pilots?

.1 U→ x: pc→m	returns	3.09×10^{15} m.
EXIT	returns to the top *view* of U→.	
1 year→s /	returns	97.8×10^{6} m/s.
ENTER↑	pushes the result on the *stack*.	
CONST c	recalls *c* =	$300.\times10^{6}$ m/s.
/	returns	$326.\times10^{-3}$ = 32.6% of *c*.

Since $1\ h = 3600\ s$ and $1\ km = 1000\ m$, you can see directly that

$$3.6\ \frac{km}{h} = \frac{3600\ m}{60 \times 60\ s} = 1\ \frac{m}{s}$$

Thus, R↓ **3.6** × returns $352.\times10^{6}$ km/h.

This corresponds to

1000 × U→ x: m→nmi. $190.\times10^{6}$ nmi/h

or 190 *megaknots*.[187]

[187] Sounds like a unit created for *Alexander the Great* visiting *Gordion* in 333 BC.

Supported by your *WP 43S*, you will find further easy ways to produce whatever conversions you may need personally in addition.

In cases of emergencies of a particular kind, it may be helpful knowing *becquerel* (Bq) equals *hertz* in your *Geiger-Müller* counter, *gray* (Gy) is the unit for deposited or absorbed energy, and *sievert* (Sv) is *gray* times a radiation dependent dose conversion factor (≥1) for the damage caused in biological material including human bodies.[188] Remember also the example on pp. 91ff.

In this field, some outdated units may be found in older literature as well:

- Pour les fidèles amis de Madame *Marie Skłodowska Curie* (1903 Nobel laureate in physics and 1911 in chemistry), there was a unit *curie* with $1\ \mathrm{Ci} = 3{,}7 \cdot 10^{10}\ \mathrm{Bq} = 3{,}7 \cdot 10^{10}\ {}^{decays}/_{\mathrm{s}}$. You can deduct from this unit that larger pieces of radioactive material were 'absolutely no problem' for the pioneers in this field.[189]
- For those admiring the very first (1901) Nobel laureate in physics, *Wilhelm Conrad Röntgen*, for discovering the X-rays (ruining his hands in those experiments since he could not know better yet), the charge generated by radiation in matter was measured by the unit *roentgen* ($1\ R = 2{,}58 \cdot 10^{-4}\ {}^{\mathrm{A\,s}}/_{\mathrm{kg}}$).[190]
- A few decades ago, *rem* (i.e. *roentgen equivalent in men*[191]) measured what *sievert* does today (1 rem = 10 mSv).
- And 1 Gy = 100 rad (i.e. *radiation absorbed dose*), which is pretty much since there is almost nothing greater than *millirad* in literature.

---

[188] Our warmest regards go to Algeria*, Australia*, Belarus*, Canada, China, France, India, Japan, Kazakhstan*, Kiribati*, the Marshall Islands* (e.g. Bikini, Eniwetok), North Korea, Pakistan, Russia, the Ukraine*, the United Kingdom, the USA, and Xinkiang-Uigur* (in alphabetical order) so far. The countries marked with a star suffer from actions of their respective 'mother states' at those times (the task to find out about those colonialists is left for the reader). The states without marks controlled their industry and/or military in a way that activated areas within their own territory could happen, too. After all, mankind gathers experience with radioactivity.

[189] *Marie Curie* died from aplastic anemia, aged 66.

[190] *Conrad Röntgen* died from carcinoma of the intestine, aged 77.

[191] This unit must be outdated – it is not regarded gender equitable nowadays anymore.

SECTION 6: CREATING YOUR VERY PERSONAL *WP 43S*

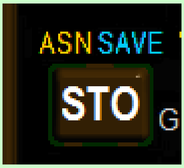
Your *WP 43S* is the first calculator worldwide allowing for fully customizing the user interface; i.e. you may assign an arbitrary function to almost any location, unshifted or shifted, on the keyboard or in a *menu*. *User mode* will then bring your personal assignments to the front, so you can interact via a user interface you designed yourself.

Even before doing such soft assignments, there are two keyboard variants supported taking care of the demands of people living in different 'mathematical regions'. The keys for multiplication, division, and the radix mark may be labelled according to your preferences:

	Default	Alternative
Division	/	: [192]
Multiplication	×	·
Radix mark	.	,

Note this manual prints the default key labels throughout its text.

Beyond these variants, use ASSIGN (ASN) for storing your personal favorite assignments. It allows for reassigning the entire keyboard except the top row of keys – these will stay *softkeys* always. Keep basic functionalities accessible (see p. 292 for some caveats).



In the explanations starting overleaf,

- ▭ stands for the *softkey* applicable (optionally headed by a *prefix*),

[192] You also find ÷ on calculators frequently. Though *ISO 80000-2* unambiguously states: "The symbol ÷ should not be used." Thus, this label is not supported on the *WP43S*.

- **[*key*]** represents an arbitrary key of your *WP 43S* (optionally headed by a *prefix*),
- ***menu*** is the name of an arbitrary *menu* defined, either
  - picked from CATALOG by entering f CATALOG MENUS, browsing to the target *menu*, and pressing the respective ▭, or
  - called from the keyboard by pressing the respective key headed by the associated *prefix*, if applicable; and
- ***name*** is the name of an arbitrary *item*. Remember an *item* may be an operation, function, digit, character, routine (label), variable, *system flag*, or a (*sub*) *menu* defined. **The name of an *item* consists of up to seven characters and must be unique within its set.** There are two sets:
  - One contains all operations, functions, constants, global labels, and (*sub*) *menus* defined at execution time of ASSIGN.
  - The other set contains all the variables and *system flags* – a variable undefined at execution time of ASSIGN will be created (as explained on p. 61).

  Note upper and lower case letters are checked, so the system will regard **Menu1** and **MENU1** as being different names. Super-scripts and subscripts are not discriminated from normal characters, so e.g. **data1T** and **data**$_1^T$ are interpreted as the same name by your *WP 43S* but the latter may ease reading for you. Where a name is required, it may be either
  - picked from CATALOG by entering f CATALOG, choosing the respective branch, browsing to the target *item*, and pressing the respective ▭, or
  - called from the keyboard by pressing the respective key (optionally headed by a *prefix*).

  Just pressing ENTER↑ where the operating system expects a name of an *item* is interpreted as input of an *empty name* and will delete the user assignment of the respective location.

Assigning Your Favourite Functions

Now here is how you can tailor the surface of your *WP 43S* according to your individual preferences:

ASN ***name*** **[*key*]**

will assign that named *item* to **[*key*]** in *user mode*. It will throw an error if said ***name*** does not exist.

Note that ASN ***name*** ☐ will assign that named *item* to the respective position in the bottom *menu* row displayed at the time you press this ☐, overwriting the label shown there. In full analogy, ASN ***name*** f ☐ and ASN ***name*** g ☐ assign said *item* to the corresponding position in the respective shifted *menu* row.

Each user assignment will hold until it is overwritten or ENTER↑ is entered for ***name*** (see above).

Note all user assignments will be accessible in *user mode* only (see pp. 292ff) **– except the *items* assigned to top row of keys in two user *menus*** (see MyMenu and Myα below): they will be displayed as long as no other *menu* is called.

Example 1:

Let's assign the statistical sample standard error to g + CC (this location is assigned to ∡ in *startup default*). There are three different ways to do this (specified here printing all keystrokes necessary):

1) f ASN f CATALOG FCNS S M s_m g CC
   This way will be demonstrated step by step starting overleaf.

2) f ASN f STAT s_m g CC

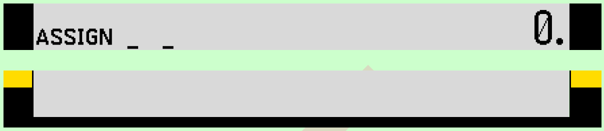
On the other hand,

f ASN ENTER↑ g CC

will reset g-shifted CC to factory default ∡ as explained at the very end of last chapter.

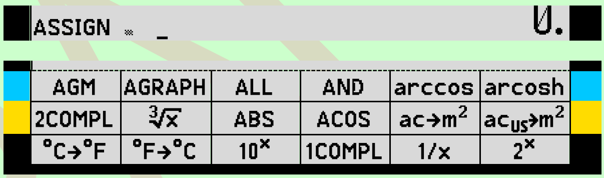
We will walk you through solution 1 step by step here, starting with a clear *stack* (press **0** FILL if necessary). Only the *menu section* and the command echo row will be shown in the following since all action will take place there:

ASN



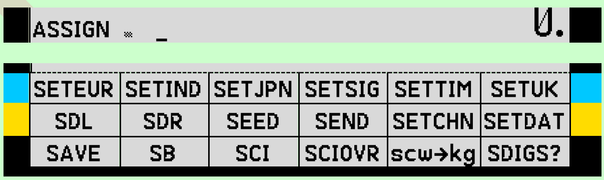
CATALOG

FCNS



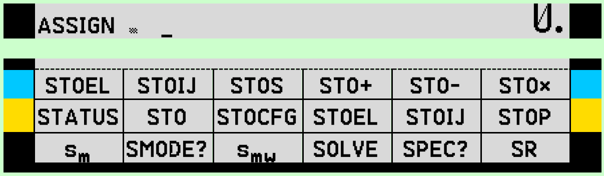
This is the top *view* of the FCNS *submenu* in CATALOG. Now enter the 1st letter of the requested command:

S



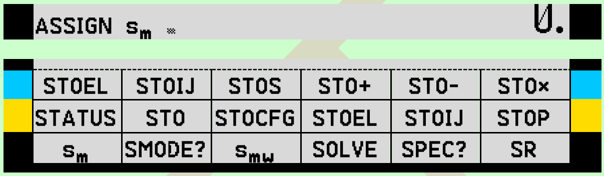
Quickly entering the 2nd letter helps significantly:

M (if you find you waited too long before pressing M, just wait another few *seconds*, then key in S M quickly here instead)

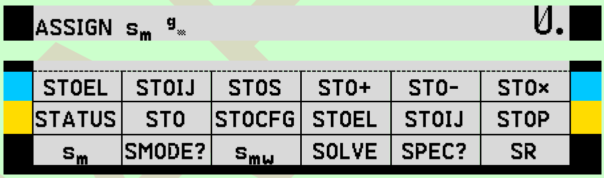


Now, press the leftmost ▭ for the function to be assigned:

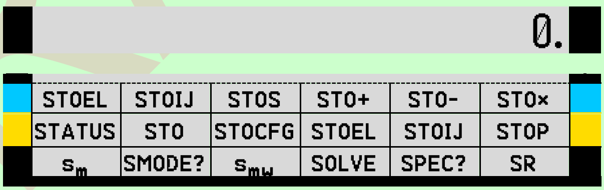
sm



g



CC



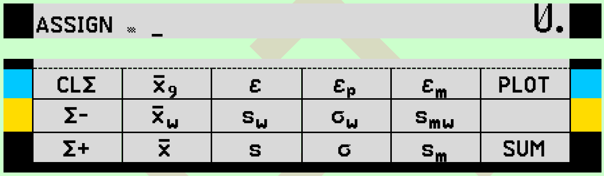
… and the assignment is done. Note this last *menu view* will stay on screen until another *view* or *menu* is called or this *menu* is EXITed explicitly. And the function ∡ will stay accessible also in *user mode* via CATALOG FCNS ... – or via default g ∡ when leaving *user mode*.

Example 2:

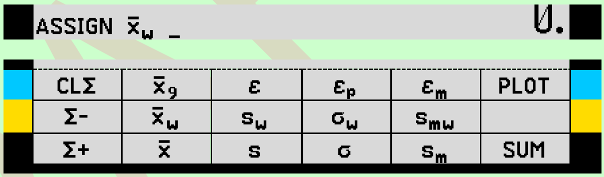
Assign the weighted arithmetic mean to the 1st key in MyMenu (assume *startup default* settings):

ASN

STAT



x̄w



USER

Note that pressing USER will exit all *menus* being open at that time so MyMenu (which is empty still) can slip on the screen.

▭ (press the leftmost *softkey*)

Summarizing,

ASN STAT x̄w USER ▭

did this assignment.

Note that MyMenu will show up whenever all other *menus* are exited completely. It will remain on screen as long as no other *menu* is called, unless your *WP 43S* is in *alpha input mode.*[193] This applies regardless whether your *WP 43S* is in *user mode* or not (see below). Thus, filling MyMenu may well be the first step of customizing your *WP 43S.* You may, for instance, put the six trigonometric functions into the unshifted row of MyMenu and will have them almost always at hand.

Creating Your Own Menus

ASN USER ***new_menu_name*** ENTER↑ will define a new user *menu*. In this sequence, ASN USER turns on *alpha input mode* so you can immediately enter the new *menu* name (up to seven characters, no blanks, and the name must be unique).

Example:

To create a *menu* **FavFun** for your favourite functions, enter:

ASN USER F ▼ A V ▲ F ▼ U N ENTER↑

The new name will be inserted in CATALOG'MENUS (ASSIGN will throw an error if the 'new' *menu* name specified will turn out being defined already). The new *menu* itself will be created with 18 blank entries – its size is fixed. You may fill it now.

Example:

Assign the y-forecasting function to the fourth key in that new user *menu* (assuming you did not define any other *menu* starting with 'Fa' before).

Also the solution of this example will be shown step by step:

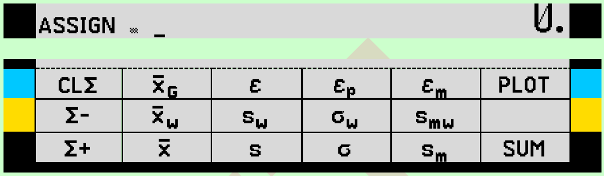
It starts with the last display of last paragraph since MyMenu stays on screen as long as no other *menu* is called, and we assigned one function to it just above.

---

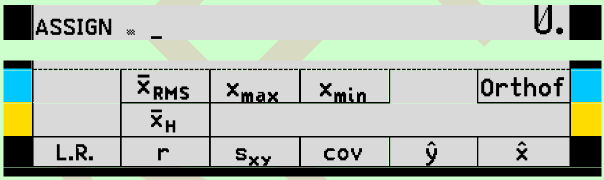
193 In *AIM*, Myα will appear instead when no other *menu* is called and will stay on screen until these conditions will change.

ASN

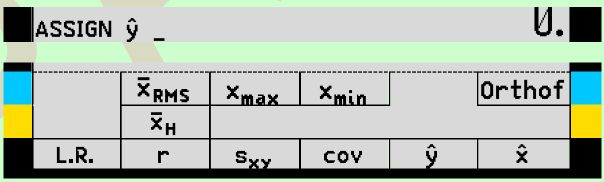
STAT



▲



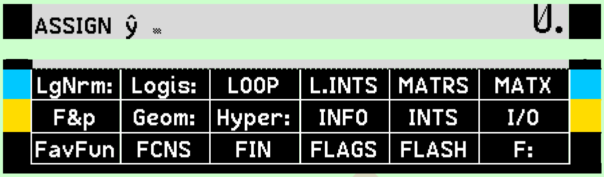
ŷ



CATALOG

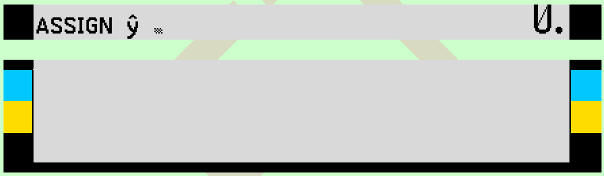
MENUS

Here you see the first *view* on all the *menus* defined on your *WP 43S*. Now, enter F and the *view* jumps to the corresponding position in this *submenu*:

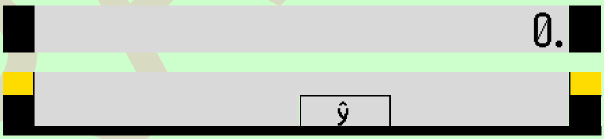


Press the leftmost *softkey*

FavFun



Since FavFun was just created above there is nothing to be seen in the *menu section* of the display yet. Pressing the fourth *softkey*, however, you will get now



Summarizing,

ASN STAT ▲ ŷ CATALOG MENUS F FavFun ▭

did the job here. Note that FavFun will remain on screen until another *menu* is called or it is EXITed explicitly.

Browsing and Purging Menus, Variables, and Programs

As seen in last paragraph,

CATALOG'MENUS contains all the *menus* currently defined. Thus,

CATALOG MENUS … ▭ ⬅ allows for deleting the *menu* selected. Predefined *menus* cannot be deleted.

Variables and programs are handled in full analogy:

CATALOG'VARS contains all variables currently defined. Thus,

CATALOG VARS … ▭ ⬅ allows for deleting the variable selected. Predefined variables cannot be deleted.

New programs must start with a global label. Such labels may be up to seven characters long and must be unique (cf. p. 282).

CATALOG'PROGS contains all programs currently defined. Thus,

CATALOG PROGS … ▭ ⬅ allows for deleting the program selected (cf. CLP and CLPALL).

Assigning Special Characters

You must be in *alpha input mode* (*AIM*) to do the following. Then,

ASN ***character*** **[*key*]** will assign the character specified to **[*key*]**. You can pick the character to be assigned from the alpha keyboard or an arbitrary alpha *menu* as introduced above (on p. 194). **[*key*]** may be any legal label location, shifted or unshifted, except USER, ENTER↑, or EXIT. The assignment will become valid when *AIM* is called in *user mode* or when *user mode* is called in *AIM*.

Example 1:

Let's assign the parentheses to f + TRI and f + ln (these locations are not assigned yet in *AIM*). Remember g − calls αMATH in *AIM* – see the *ReM* for its contents.

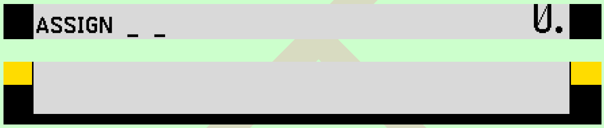
α ASN g − f (f TRI

ASN f) f ln

Example 2:

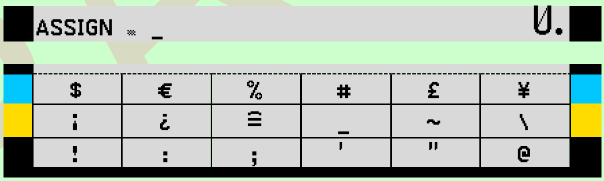
Assign the *Yuan* symbol ¥ (contained in α• at a g-shifted position) to the 1st key in Myα (assume *startup default* settings once again):

α ASN

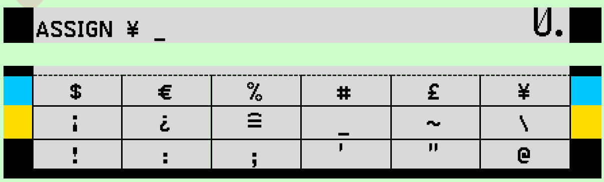


g

.



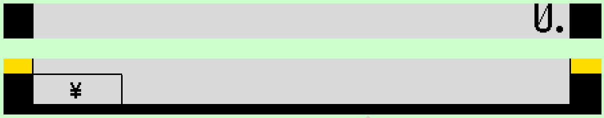
g ¥ (press the rightmost *softkey*)



USER

Note that USER will exit all *menus* being open at that time so Myα can slip on the screen (being empty still).

▢ (press the leftmost *softkey*)



Summarizing,

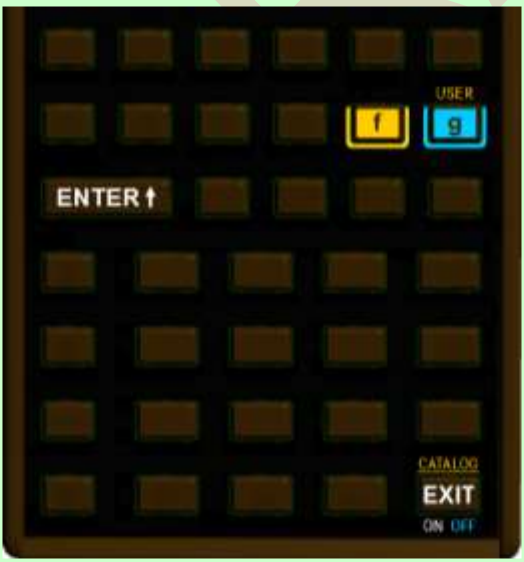
ASN g . ¥ USER ▢

did this assignment.

User Mode

USER toggles *user mode*. Therein, your (user) assignments become valid wherever they apply. Everything is wide open for your ideas except the top row of keys (being controlled by MyMenu and Myα, cf. pp. 286ff).

User mode gives you unexcelled freedom for creating your personal calculator layout and user interface. Enjoy – and play with the opportunities you have. For obvious reasons, we recommend leaving f, g, ENTER↑, CATALOG, EXIT / ON, and OFF untouched (note EXIT and ON are connected). And do not forget you will need USER for returning from *user mode*.



WARNING: Do not remove inevitably necessary functionalities from the keyboard by assigning. In case of emergency, a hard reset will be your only escape – erasing all your precious programs and data but those you saved in *flash memory*. Thus, checking all consequences meticulously before assigning functions is highly recommended; please note **all your assignments are strictly at your own risk**.

Pressing any function key (or a *prefix* plus a key) displays a preview of the operation currently assigned to it left in the T numeric row – if you realize you have picked the wrong key, simply keep it pressed until the display falls back to NOP after 1 *second*.[194] This preview is particularly helpful in *user mode*, when the function executed by a key may not be the one indicated on the keyboard.

Once you have reached a stable user layout, we recommend storing it (using STOCFG) in a *register* or variable, together with the other settings mentioned on p. 80. This applies especially if you plan having further alternative layouts – you can load any of them using RCLCFG.[195] Think e.g. of storing a dedicated set of assignments for working with *short integers* featuring *Boole's* operations as primary functions.

Printing keyboard overlays for your favorite layouts may pay well,

---

[194] Preview and fallback apply for all key functionalities except 0 … 9, ., E, ←, and EXIT (and +/− in numeric entry). f and g are echoed but will not fall back. (On the *HP-42S*, preview and fallback are absent also for PRGM, ASSIGN, STO, RCL, XEQ, SHOW, and OFF.)

[195] RCLCFG will throw an error if you try recalling something different than a *configuration*.

especially if you reassign just functions printed on the key plate. Overlays cover this plate entirely (see the *ReM*, *App. F*, for their dimensions) and are fixed in the slots provided on either side of the keyboard.

Should you get lost in your various user assignments, however, look for U top right in the *status bar* – and remember that pressing USER will return immediately from *user mode* to the factory default keyboard of your *WP 43S* as you know it from the very beginning. And if you want to get rid of outdated user layouts and free the memory allocated for the respective assignments, simply clear the respective *registers* or delete the allocated variables as described on p. 290.

We sign off wishing you long lasting joy and benefit working with your very own, personalized *WP 43S*!

APPENDIX 1: OPERATOR PRECEDENCE

Your *WP 43S* does not have to care for operator precedence since it executes just one operation at a time (cf. p. 46). Hence it is your job to control the sequence of operations you present to your *WP 43S.* There are common rules and conventions in mathematics dealing with that – you have learned them in school. Here is just one **example** for affirmation and/or reminding:

$$1 - 2 \cdot 3^4 : 5 + \sin\left(6 - \sqrt[3]{7^2}\right) \cdot 8! + ln\left[\left(-9^{2^3} \cdot 45^{(6/7)}\right)^2\right]$$

(or, written for another part of this world needing more space:

$$1 - 2 \times 3^4 \div 5 + \sin\left(6 - \sqrt[3]{7^2}\right) \times 8! + ln\left[\left(-9^{2^3} \times 45^{(6/7)}\right)^2\right])$$

This may be solved the following way, for instance, using your *WP 43S* with *startup default* settings:

9 [ENTER↑] 2 [ENTER↑] 3 [y^x] [y^x] [+/-]	calculates -9^{2^3} ; note the arguments automatically fill in correctly.
6 [ENTER↑] 7 [/] 45 [x⇄y] [y^x]	calculates $45^{(6/7)}$.
[×] [x^2] [ln x]	solves the rightmost term.
7 [x^2] 3 [1/x] [y^x] 6 [x⇄y] [−] [sin]	solves the sine.
8 [x!] [×] [+]	solves the 3rd term and adds it to the fourth.
3 [ENTER↑] 4 [y^x] 2 [×] 5 [/] [−]	solves the 2nd term and subtracts it from said sum.
1 [+] returns the overall solution	1 657.008 948 091 604

The colors indicate the three *stack registers* employed for this solution (cf. pp. 41ff). Note [x⇄y] is used twice herein to swap arguments.

APPENDIX 2: KEY RESPONSE TABLE

Here you find all direct keystroke inputs explained, top left to bottom right of the keyboard. For each key, its unshifted function is mentioned first, then its f-shifted and its g-shifted function, if applicable.

Most keys will change functionality in *alpha input mode* (*AIM*), hence the “alpha” meanings are listed thereafter. See the pages mentioned explicitly or the *ReM* for details of all the functions mentioned below.

<table>
<tr><th>R</th><th>Keystrokes</th><th colspan="2">Meaning</th></tr>
<tr><td rowspan="3">1</td><td>[ ]</td><td>Calls the function displayed at the corresponding position in the bottom softkey row of the LCD.</td><td rowspan="3">Does nothing if there is no function displayed at this position. Cf. pp. 27f.</td></tr>
<tr><td>f [ ]</td><td rowspan="2">Call the function displayed at the corresponding position in the golden or blue softkey row, respectively.</td></tr>
<tr><td>g [ ]</td></tr>
<tr><td rowspan="6">2</td><td>1/x</td><td colspan="2">Inverts the number x or all elements of the matrix x.</td></tr>
<tr><td>a b/c</td><td colspan="2">Enters fraction display mode (FDM), i.e. displays all reals as proper fractions or mixed numbers. If FDM was active already, toggles display between proper and improper fractions.</td></tr>
<tr><td>α.FN</td><td colspan="2">Opens the menu of operations for alpha string manipulation. Cf. pp. 198f.</td></tr>
<tr><td>A</td><td>Enters the letter A or a</td><td rowspan="3">in AIM (cf. pp. 193ff).</td></tr>
<tr><td>f A</td><td>Opens the catalog of all Latin letters provided (also accented ones)</td></tr>
<tr><td>g A</td><td>Enters the Greek letter Α or α</td></tr>
</table>

R	Keystrokes	Meaning
2	y^x	Raises y to the power of x.
	#	If pressed trailing integer input, defines its base. Else converts x into a *short integer* of the base specified, cf. pp. 136ff.
	EXP	Opens a *menu* containing x^3, roots, 2^x, logarithms, hyperbolic and some exponential functions more (cf. p. 27).
	B	Enters the letter **B** or **b** in *AIM* (cf. pp. 193ff).
	#	Enters the character # in *AIM* (cf. pp. 193ff).
	g B	Enters the Greek letter Β or β in *AIM* (cf. pp. 193ff).
2	TRI	Opens the *menu* containing SIN, COS, TAN, hyperbolic functions, and their inverses (cf. p. 29)..
	d.ms	If pressed trailing numeric input, enters an *angle* in *degrees*, *minutes*, and *seconds* (i.e. sexagesimal notation). Else sets *angular display mode* to sexagesimal angles. Cf. pp. 125ff.
	π	Recalls the number π into **X**.
	C	Enters the letter **C** or **c** in *AIM*.
	g C	Enters the Greek letter Γ or γ in *AIM*.
2	ln	Returns the natural logarithm of x.[196]
	.d	If pressed trailing numeric input, enters a *date* (cf. p. 191). Else leaves fraction display mode (see a b/c above) and converts • an integer to a *real number* (cf. p. 135), • a sexagesimal *angle* to a decimal number (cf. p. 128), • a sexagesimal *time* to a decimal number (cf. p. 190).
	lg	Returns the (common) decadic logarithm of x. [196]
	D	Enters the letter **D** or **d** in *AIM*.
	g D	Enters the Greek letter Δ or δ in *AIM*.

[196] I.e. either of the number x or of all elements of the matrix x.

<table>
<tr><th>R</th><th>Keystrokes</th><th colspan="2">Meaning</th></tr>
<tr><td rowspan="5">2</td><td>e^x</td><td colspan="2">Raises e to the power of x. [196]</td></tr>
<tr><td>h.ms</td><td colspan="2">If pressed trailing numeric input, enters a sexagesimal time. Else converts x to such a time. Cf. pp. 189f.</td></tr>
<tr><td>10^x</td><td colspan="2">Raises 10 to the power of x. [196]</td></tr>
<tr><td>E</td><td>Enters the letter E or e</td><td rowspan="2">in AIM (cf. pp. 193ff).</td></tr>
<tr><td>g E</td><td>Enters the Greek letter Ε or ε</td></tr>
<tr><td rowspan="6">2</td><td>x^2</td><td colspan="2">Returns the square of x. [196]</td></tr>
<tr><td>α</td><td colspan="2">Sets AIM for entering characters (cf. pp. 193ff).</td></tr>
<tr><td>$\sqrt{x}$</td><td colspan="2">Extracts the square root of x. [196]</td></tr>
<tr><td>F</td><td>Enters the letter F or f</td><td rowspan="3">in AIM.</td></tr>
<tr><td>f x^2</td><td>Enters the character √</td></tr>
<tr><td>g F</td><td>Enters the Greek letter Φ or φ</td></tr>
<tr><td rowspan="5">3</td><td>STO</td><td colspan="2">Stores (copies) x in the destination specified (cf. pp. 53ff).</td></tr>
<tr><td>ASN</td><td colspan="2">Assigns an item to a key, allowing you to create your very personal user keyboard layout (cf. pp. 281ff).</td></tr>
<tr><td>SAVE</td><td colspan="2">Saves all your data in the backup region (cf. p. 234) of FM from where they may be recovered by LOAD entirely.</td></tr>
<tr><td>G</td><td>Enters the letter G or g</td><td rowspan="2">in AIM.</td></tr>
<tr><td>g G</td><td>Enters the Greek letter Γ or γ</td></tr>
</table>

R	Keystrokes	Meaning
3	RCL	Recalls (copies) a stored object into **X** (cf. pp. 53ff). – If pressed in RBR, leaves RBR after recalling the object at the bottom line or entering a corresponding step (cf. pp. 261ff).
	RBR	Calls the *register browser* (cf. pp. 261ff).
	VIEW	Views the destination, i.e. displays its address and contents directly below the *status bar* until next keystroke (cf. p. 59).
	H	Enters the letter **H** or **h** in *AIM* (cf. pp. 193ff).
	g H	Enters the Greek letter Χ or χ in *AIM* (cf. pp. 193ff).
3	R↓	Rolls the *stack* contents one level down (cf. p. 39).
	R↑	Rolls the *stack* contents one level up.
	CPX	Opens the *menu* of commands operating on *complex numbers* like CONJ, CROSS, DOT, and Re⇄Im. Cf. pp. 154ff.
	I	Enters the letter **I** or **i** in *AIM*.
	f R↓	Makes next character a subscript (if applicable) in *AIM*.
	g I	Enters the Greek letter Ι or ι in *AIM*.
3	CC	*Complex* closing, composing, cutting, and converting, see pp. 154ff and 307.
	|x|	Returns the absolute (unsigned) value of ***x***. [196]
	∡	Either returns the phase of ***x*** [196] or the angle between the vectors ***x*** and ***y***.
	J	Enters the letter **J** or **j** in *AIM*.
	g J	Enters the Greek letter Η or η in *AIM*.
3	f	*Prefix* to reach a secondary gold function label. Pressing f twice will clear this *prefix*.
	SNAP	Dumps the current screen to a file on the calculator's USB flash drive.

R	Keystrokes	Meaning
3	g	*Prefix* to reach a secondary blue function label. Pressing g twice will clear this *prefix*.
	USER	Toggles *user mode* (see pp. 292ff).
4	ENTER↑	Context sensitive key, see p. 307.
	STATUS	Returns free space available, memory currently used, *user* and *system flags* set (cf. pp. 263f).
	DROP↓	Drops ***x*** from the *stack* (cf. p. 39).
4	x⪌y	Swaps the contents of **X** and **Y** (cf. p. 39).
	FILL	Fills all *stack registers* with ***x*** (cf. p. 39).
	STK	Opens the *menu* of *stack* related operations (drop, swap, and shuffle commands). Cf. pp. 38ff.
	K	Enters the letter **K** or **k** in *AIM* (cf. pp. 193ff).
	f x⪌y	Enters the character ⪌ in *AIM* (cf. pp. 193ff).
	g K	Enters the Greek letter **Κ** or **κ** in *AIM* (cf. pp. 193ff).
4	+/_	If pressed during input of mantissa or exponent, changes its sign (cf. p. 25). Else multiplies ***x*** times −1.
	Δ%	Returns ***x*** – ***y*** % of ***y***. Leaves ***y*** unchanged.
	FIN	Opens the *menu* of financial functions (i.e. % functions and the application TVM – see pp. 266ff and 310ff).
	L	Enters the letter **L** or **l** in *AIM*.
	f +/_	Enters the character **±** in *AIM*.
	g L	Enters the Greek letter **Λ** or **λ** in *AIM*.

R	Keystrokes	Meaning	
4	E	Allows entering an exponent of ten for convenient entry of very large or very small numbers (cf. p. 25).	
	SHOW	Shows the number ***x*** with its maximum precision until next keystroke.	
	DISP	Opens a *menu* containing FIX, SCI, ENG, and more commands for numeric display formatting. Cf. pp. 80ff.	
	M	Enters the letter **M** or **m**	in *AIM* (cf. pp. 193ff).
	f E	Makes next character a superscript (if applicable)	
	g M	Enters the Greek letter Μ or μ	
4	←	Context sensitive key, see p. 309.	
	↶	Undoes the last command executed (cf. p. 51).	
	CLR	Calls a *menu* containing commands for clearing; cf. p. 52.	
5	/	Divides ***y*** by ***x***. For matrices, multiplies ***y*** times x^{-1}.	
	RMD	Returns the remainder of ***y*** divided by ***x***.	
	MOD	Returns ***y*** modulo ***x***.	
	N	Enters the letter **N** or **n**	in *AIM*.
	f /	Enters the character **/**	
	g N	Enters the Greek letter Ν or ν	
5	7	If there is an open question like `Are you sure?`, enters **N** for 'no'. Else enters the digit **7**.	
	CONST	Opens a catalog of fundamental physical, mathematical, astronomical, and surveying constants. Cf. pp. 270ff.	
	O	Enters the letter **O** or **o**	in *AIM*.
	f 7	Enters the character **7**	
	g O	Enters the Greek letter Ω or ω	

R	Keystrokes	Meaning
5	8	Enters the digit **8**.
	P	Enters the letter **P** or **p**
	f 8	Enters the character **8** in *AIM* (cf. pp. 193ff).
	g P	Enters the Greek letter Π or π
5	9	Enters the digit **9**.
	RTN	Returns to the caller. Cf. pp. 202ff.
	Q	Enters the letter **Q** or **q** in *AIM*.
	f 9	Enters the character **9**
5	XEQ	If there is an open question like `Are you sure?`, confirms it; else – if in *PEM* – inserts a call to the subroutine with the label specified; else (i.e. in *run mode*) calls the routine with the label specified and starts executing it.
	GTO	Goes to the specified location in program memory.
	LBL	Enters a label for a particular location in program memory.
6	×	Multiplies y times x.
	x!	Returns the factorial of x (or $\Gamma(x+1)$ for non-integer x).
	PROB	Opens a *menu* containing combinations, permutations, the Gamma function, a random number generator, and all probability distributions supported. Cf. pp. 96ff.
	R	Enters the letter **R** or **r**
	f ×	Enters the character **×** or **·** in *AIM*.
	g R	Enters the Greek letter Ρ or ρ

R	Keystrokes	Meaning
6	4	Enters the digit **4**.
	STAT	Opens the *menu* of sample statistics operations: Σ+, Σ−, CLΣ, various means and measures for scattering, as well as curve fitting functions and settings. Cf. pp. 99ff.
	Σ	Opens the *menu* of accumulated statistical sums, cf. p. 118.
	S	Enters the letter **S** or **s**
	f 4	Enters the character **4** in *AIM* (cf. pp. 193ff).
	g S	Enters the Greek letter **Σ** or **σ**
6	5	Enters the digit **5**.
	R←	Calls →REC, converting polar coordinates ***r*** (in **X**) and ***ϑ*** (in **Y**) to rectangular (*Cartesian*) coordinates ***x*** and ***y*** (cf. p. 128).
	→P	Calls →POL, converting rectangular coordinates (***x*** and ***y***) to polar coordinates ***r*** (in **X**) and ***ϑ*** (in **Y**, cf. pp. 20f).
	T	Enters the letter **T** or **t**
	f 5	Enters the character **5** in *AIM*.
	g T	Enters the Greek letter **Τ** or **τ**
6	6	Enters the digit **6**.
	U→	Opens the *menu* of unit conversions. Cf. pp. 276ff.
	L→	Opens the *menu* of angular conversions. Cf. p. 126.
	U	Enters the letter **U** or **u**
	f 6	Enters the character **6** in *AIM*.
	g U	Enters the Greek letter **Θ** or **ϑ**
6	▲	Context sensitive key, see p. 309.
	≡△	Moves the program pointer one step back. Cf. pp. 202ff.
	FLAGS	Opens the *menu* of *flag* commands. These are of most use in *PEM*. Cf. pp. 202ff.

R	Keystrokes	Meaning	
7	−	Subtracts x from y.	
	ADV	Opens a *menu* of advanced operations for solving arbitrary equations, finding roots, integrating, deriving, computing sums and products (cf. pp. 235ff).	
	EQN	Opens the *menu* of all equations currently defined (cf. pp. 238ff).	
	V	Enters the letter **V** or **v**	in *AIM* (cf. pp. 193ff).
	f −	Enters the character **–**	
	g −	Opens a *menu* of math symbols	
7	1	Enters the digit **1**.	
	BITS	Opens a *menu* containing Boole's operations (AND, OR, NOT, etc.) as well as bit manipulating commands.	Both *menus* are most useful with *short integers*, cf. pp. 136 and 140ff.
	INTS	Opens a *menu* of operations for integers as well as sign mode settings.	
	W	Enters the letter **W** or **w**	in *AIM*.
	g W	Enters the Greek letter Ψ or ψ	
7	2	Enters the digit **2**.	
	MATX	Opens the *menu* of matrix operations including e.g. $[M]^{-1}$, |M|, $[M]^{T}$, CROSS, DOT, and the *Matrix Editor* (cf. pp. 163ff).	
	X.FN	Opens a *menu* of advanced mathematical (extra) functions. See the *ReM*.	
	X	Enters the letter **X** or **x**	in *AIM*.
	f 2	Enters the character **2**	
	g X	Enters the Greek letter Ξ or ξ	

R	Keystrokes	Meaning
7	[3]	If there is an open question like `Are you sure?`, enters **Y** for 'yes'. Else enters the digit **3**.
	TIMER	Calls the timer application (cf. pp. 264ff).
	CLK	Opens the menu of *time* and *date* commands. Cf. pp. 189ff.
	[Y]	Enters the letter **Y** or **y**
	[f] [3]	Enters the character **3** — in *AIM* (cf. pp. 193ff).
	[g] [Y]	Enters the Greek letter **Υ** or **υ**
7	[▼]	Context sensitive key, see p. 309.
	≡▽	Moves the program pointer one step forward (cf. pp. 202ff).
	MODE	Opens a *menu* of operations for setting modes like angular display format, max. denominator, etc. (cf. pp. 125ff and 151ff).
8	[+]	Adds x to y.
	I/O	Opens the *menu* of I/O-related operations. Cf. pp. 233f.
	PRINT	Opens the *menu* of print-related operations.
	[Z]	Enters the letter **Z** or **z**
	[f] [+]	Enters the character **+** — in *AIM*.
	[g] [Z]	Enters the Greek letter **Ζ** or **ζ**
8	[0]	Enters the digit **0**.
	LOOP	Opens a *menu* containing INC and DEC and the related loop control commands ISG, DSE, etc. Cf. pp. 218f.
	TEST	Opens the *menu* of comparisons, conditionals, and other binary tests. Cf. pp. 214ff.
	[?]	Enters the character **?**
	[f] [0]	Enters the character **0** — in *AIM*.
	[g] [0]	Enters the printer character 🖶

R	Keystrokes	Meaning	
8	.	Usually enters a decimal radix mark in numeric input. If pressed twice in numeric input, allows for entering a fraction (cf. pp. 68f and 151ff). In *register* or *flag* addressing, . heads a local address (cf. pp. 57ff).	
	PARTS	Opens a *menu* containing FP, IP, SIGN, DECOMP, etc.	These operations are most useful in *PEM*. Cf. pp. 202ff.
	INFO	Opens a *menu* of commands to return system information. Cf. p. 217.	
	,	Enters a comma	in *AIM* (cf. pp. 193ff)
	f .	Enters a point	
	g .	Opens a *menu* of punctuation marks etc.	
8	R/S	Context sensitive key, see p. 308.	
	P/R	Toggles *program-entry* and *run mode*.	
	P.FN	Opens a *menu* of dedicated programming functions. These are of most use in *PEM*. Cf. pp. 202ff.	
	␣	Enters a blank space in *AIM*.	
8	EXIT / ON	Context sensitive key, see p. 308.	
	CATALOG	Opens the *catalog* of everything (functions, variables, menus, programs, etc.). See the *ReM* for its structure and contents.	
	OFF	Turns your *WP 43S* off unless in *PEM*, where it inserts OFF behind the *current step* (cf. p. 204).	

Seven context sensitive keys need longer explanations – find them in the table below, sorted alphabetically. If any of these keys is pressed, your *WP 43S* will run top down through a sequence of key-specific tests – whichever test becomes true first, your *WP 43S* will execute the corresponding operation and return, waiting for next input.

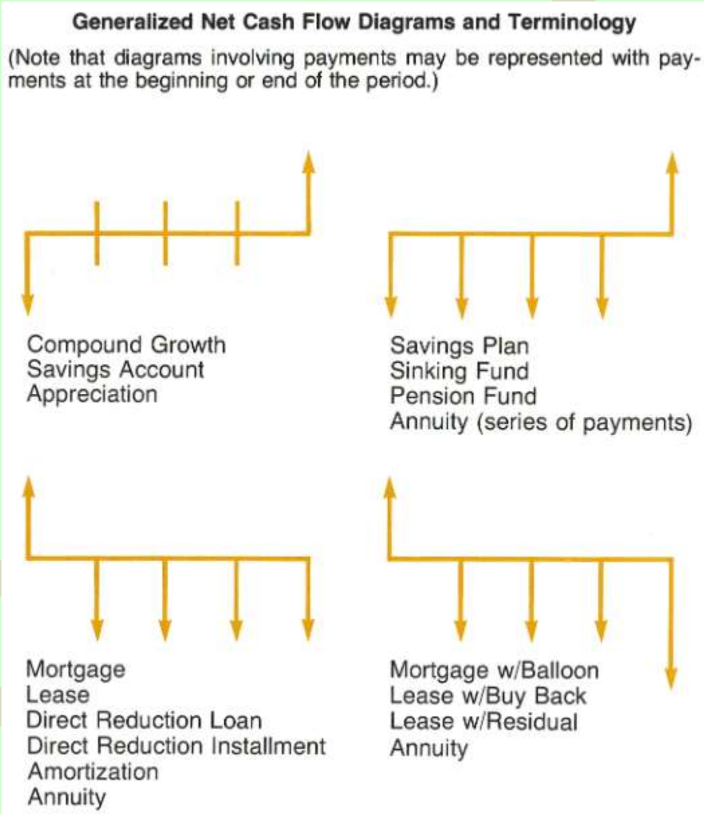
Key	Condition(s)	Meaning
CC	**X** contains an <u>open</u> (input) number, cf. p. 25	If POLAR is clear, CC closes input, checks, and saves it as *real* part of a forthcoming *complex number*, then waiting for your input of its *imaginary* part. Else CC closes, checks, and saves the input as magnitude and waits for your input of the phase. Cf. pp. 154ff for more.
	X contains a closed *complex number*, vector, or matrix	If POLAR is clear, CC splits ('cuts') x into its *real* and *imaginary* part, returning the *real* part in **Y** and the *imaginary* part in **X**. Else CC splits x into its magnitude r and phase ϑ, returning r in **Y** and ϑ in **X**.
	X and **Y** contain two closed *reals*	Interprets y and x either (for POLAR set) as magnitude and phase, or (for POLAR clear) as *real* and *imaginary* parts. CC combines y and x to compose one *complex number* x, then drops y.
	X and **Y** contain two closed *real* vectors (or matrices) of identical dimension	Returns one *complex* vector (or matrix) x, working in analogy to previous row.
	Else	Throws an error.
ENTER↑	Waiting for parameter input	Closes pending command input and executes said command (cf. p. 63 for more).
	Asking for confirmation	Confirms the question.
	In TIMER	Is honored as described on pp. 264f.
	In RBR, STATUS	Does nothing.
	Else	Closes alphanumeric input and enters data in the *stack* (cf. pp. 33f and 39 for details).

Key	Condition(s)	Meaning
EXIT / ON	*WP 43S* turned off	Works as ON turning your *WP 43S* on.
	Waiting for parameter input	Cancels the pending command.
	Waiting for alpha-numeric input	Closes input (note alphanumeric includes numeric input).
	Temporary information displayed	Clears this information (e.g. an error message) returning to the calculator state as was before it was thrown. Cf. p. 68.
	Asking for confirmation	Denies the question.
	In RBR, STATUS, TIMER	Leaves the application (cf. pp. 261ff).
	In a (*sub*-) *menu* or *browser*	Leaves the current (*sub*-) *menu* or *browser* without executing anything, returning to the status of your *WP 43S* as it was before.
	⌛ flashing	Stops executing the running program immediately. ! will be lit until next keystroke.
	In *PEM*	Leaves *program-entry mode* like P/R.
	A or α	Closes ***x*** and leaves *alpha input mode*.
	Else	Does nothing.
R/S	In TIMER	Starts or stops the timer without changing its value (cf. pp. 264f).
	⌛ flashing	Stops executing the running program immediately. ! will be lit until next keystroke.
	In *PEM*	Enters the command STOP.
	Else	Runs the *current routine* (cf. pp. 202ff) or resumes its execution starting with the step after the *current step*.

Key	Condition(s)	Meaning
[▲] or [▼]	After STO or RCL	Honored as described on pp. 58ff.
	In RBR, STATUS, TIMER	Honored as described in *Sect. 5* (pp. 261ff).
	A & in (<u>αINTL</u> or <u>A…Ω</u>)	[▼] sets lower case.
	α & in (<u>αINTL</u> or <u>A…Ω</u>)	[▲] sets upper case.
	A else	[▼] sets lower case. Else continue testing.
	α else	[▲] sets upper case. Else continue testing.
	In <u>EQN</u>	[▲] goes to next and… [▼] to previous equation, if applicable.
	In a *multi-view menu*	[▲] goes to next and… [▼] to previous *view* in the current *menu*.
	In *PEM*	[▲] goes to previous and… [▼] to next program step. Will repeat with $2Hz$ when pressed longer than $0.5s$.
	In *run mode*	Browses the *current routine* with… [▲] going to previous program step and… [▼] executing the current program step and going to next step.
[←]	Open alphanumeric input	Deletes the last character entered. If none is left, cancels pending command like EXIT.
	Temporary information displayed	Clears the information returning to the calculator state as was before this (e.g. an error message) was thrown. See p. 68.
	Asking for confirmation	Denies the question.
	In TIMER	Resets the timer (cf. pp. 264f).
	In *PEM*	Deletes the current program step.
	Else	Calls the command CLX.

APPENDIX 3: FURTHER APPLICATIONS OF TVM

Throughout TVM pictures, amounts received a represented by arrows pointing up, money laid out (paid, invested) by arrows pointing down. Various types of financial problems can be sketched like this then:[197]



The following examples as well as all the other text printed blue in this appendix are quoted from the *HP-27 OH*. All calculations are executed in FIX 2. Enjoy the boundary conditions of that time – those were the days ...

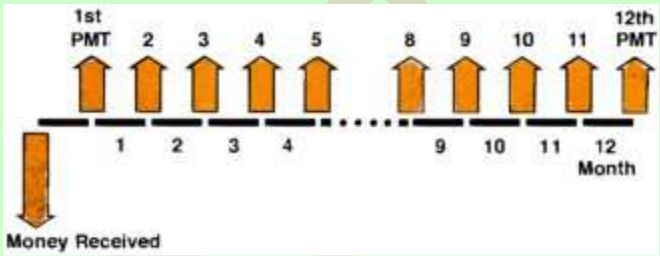
---

197 Translator's note: You can use this picture as a dictionary of some financial terms in (American) English. The word "with" is abbreviated by "w/" although this does not save any space here. Abbreviomania …

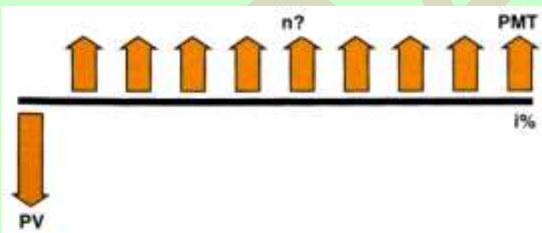
Ordinary Annuities (a.k.a. Payments in Arrears)

An *annuity* is a series of equal payments made at regular intervals. The time between annuity payments is called the payment interval or payment period. If your payment is due at the end of each payment period, it's called an *ordinary annuity* or *payment in arrears*. Examples of ordinary annuities are a car loan (where you drive away now and pay later) or a mortgage (where the payments start one month after you get your loan).

The time / money relationship for an ordinary annuity with monthly payments for a year would look like this →



Example for finding the number of periods for an ordinary annuity:



Through an insurance fund, you have accumulated $50 000 for your retirement. How long can you withdraw $3 000 every 6 months (starting 6 months from now) if the fund earns 5% per annum compounded semiannually?

Solution:

FIN TVM End		withdrawals are due at the end of each period,
5 ENTER↑ **2** / i%/a	2.50	% semiannual interest rate,
50000 PV	50 000.00	principal (capital),
3000 PMT n_{PER}	21.83	semiannual withdrawals, so your savings will last for almost 11 years.

Example 1 for finding the interest rate for an ordinary annuity:

What is the annual interest rate (a.k.a. *APR* for *annual percentage rate*) on a 2-year, $1 775 loan with $83.65 monthly payments?

Solution:

12 per/a	12.00	months per year,
12 ENTER↑ 2 × n_{PER}	24.00	periods in total,
1775 PV	1 775.00	principal (capital),
83.65 PMT	83.65	payment;
i%/a	12.11	% *APR*.

Borrowers are sometimes charged fees related to the issuance of a mortgage, which effectively raises the interest rate. Given the basis of the fee charge, the true annual percentage rate may be calculated.

Example 2 for finding the interest rate for an ordinary annuity:

A borrower is charged 2 points for the issuance of his mortgage. If the mortgage amount is $50 000 for 30 years, and the interest rate is 9% per year, with monthly payments, what annual percentage rate is the borrower paying? (1 point is equal to 1 % of the mortgage amount.)

Solution:

First, compute the payment amount which is based on $50 000

9 i%/a	9.00	annual interest rate,
12 per/a	12.00	months per year,
12 ENTER↑ 30 × n_{PER}	360.00	periods in total,
50000 PV	50 000.00	principal (capital);
PMT	83.65	payment.
PMT	83.65	reuse payment,
RCL n_{PER} n_{PER}	360.00	recall and reuse periods,
RCL PV 2 % − PV	49 000.00	effective amount received,
i%/a	12.11	% effective *APR*.

What's really happening? For a mortgage with fees, the borrower is making payments on the original loan amount, which corresponds with the initial calculation of the payment amount. If you borrow $10 000, but are immediately charged $500 in fees, you really only receive $9 500.

But, your payments are based on $10 000. With fees, then, you're really paying the same for less money, which generates the need to compute the true *APR*.

Example for finding the payment amount for an ordinary annuity:

Find the monthly payment amount on a 30-year, $52 000 mortgage at 9.75% annual interest rate.

Solution:

12 per/a	12.00	months per year,
12 ENTER↑ **30** × n_{PER}	360.00	payment periods in total,
52000 PV	52 000.00	mortgage,
9.75 i%/a	9.75	% annual interest rate;
PMT	446.76	monthly payment.

A common financial occurrence is an annuity that has a large payment at the end. The last payment – usually considerably larger although it could also be smaller than the others – is called a *balloon payment* or *balloon*.

By subtracting the present value of the balloon payment from the loan amount, the problem effectively becomes "What is the monthly payment on a direct reduction loan?"

Example (finding the payment for an ordinary annuity with *balloon*):

Yellowstone Sam is heading north, and will invest in an $8 000 dog sled and team. His loan specifies 60 monthly payments at 10% with a *balloon payment* in the 60th month of $3 000. What will his monthly payments be?

Solution:

12 per/a	12.00	months per year,
60 n_{PER}	60.00	payment periods in total,
10 i%/a	10.00	% annual interest rate;
3000 FV	3 000.00	future value of *balloon*,
PV	1 823.37	present value of *balloon*;

PV	1 823.37	input of ***PV*** of balloon;
RCL i%/a i%/a	10.00	recall & reuse interest rate,
RCL n_{PER} n_{PER}	360.00	recall and reuse periods,
8000	8 000.00	gross value of loan amount,
RCL PV − PV	6 176.63	net present value of loan amount less *balloon*;
PMT	131.24	monthly payment.

Example for finding the present value of an ordinary annuity:

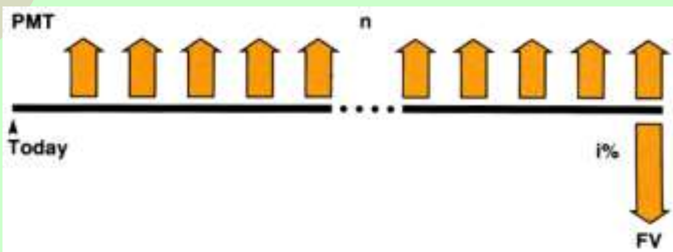
Yellowstone Sam decides to purchase a snowmobile. He plans to pay $80 per month for 3 years, and he's willing to pay 10% annual interest. How much can he afford to pay for the snowmobile?

Solution:

12 per/a	12.00	months per year,
12 ENTER↑ **3** × n_{PER}	36.00	payment periods in total,
10 i%/a	10.00	% annual interest rate;
80 PMT	9.00	monthly payment,
PV	2 479.30	price he can pay for the snowmobile.

With loan calculations, you generally solve for ***n***, ***i***, ***PMT***, or ***PV***. There is another type of ordinary annuity called a "*sinking fund*", where you make payments at regular intervals into a fund to discharge a debt (for example, to pay off a bond issue at maturity). With *sinking fund* calculations, you solve for ***n***, ***i***, ***PMT***, or ***FV*** (how much you will have in the fund at a future date).

Sinking fund payments start at the end of the first period, like so →



This is different from opening a savings account with a starting deposit today. Savings are annuity due calculations and will be described later in this section.

Example for finding the future value of an ordinary annuity:

A $100 000 bond is to be discharged by the sinking fund method. If, starting 6 months from now, you deposit $3 914.75 twice a year into a sinking fund that pays 5% compounded semiannually, will you be able to pay off the bond in 10 years?

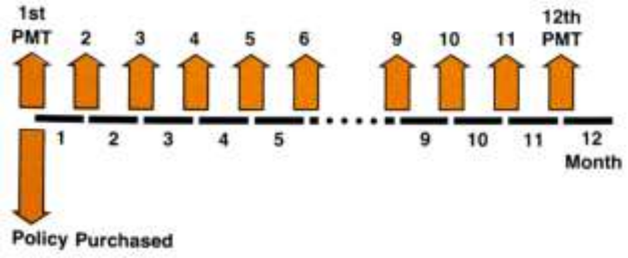
Solution:

2 per/a	2.00	halves per year,
10 ENTER↑ **2** × nPER	20.00	payment periods in total,
5 i%/a	5.00	% annual interest rate;
3914.75 PMT	3 914.75	semiannual deposit,
FV	100 000.95	balance of the fund after 10 years – it will just make it!

Annuities Due (a.k.a. Payments in Advance)

With some annuities – like insurance premiums or a lease – the payment is due at the beginning of the month. This is called an *annuity due* because the payment falls at the beginning of the payment period. Other terms are *payments in advance* or *anticipated payments*.

An annuity due with monthly payments for a year – say, a car insurance policy[198] – looks like this →



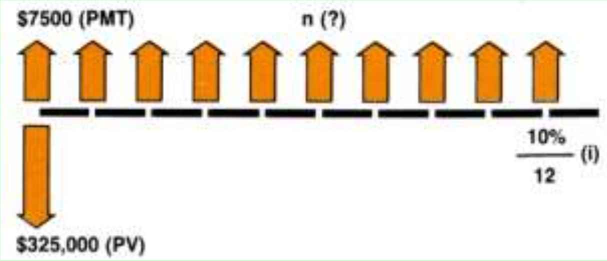
Notice that with an annuity due, you have a payment right away at the beginning of the first interval (with an ordinary annuity, your payment is not due until the end of the first period, but you also have a payment at the end of the entire term).

The following calculations all deal with *annuity due* problems, e.g. savings, insurance, leases, and rents.

---

198 Translator's note for German readers: "*Policy*" entspricht hier einer *Police*.

Example 1 for finding the number of periods for an annuity due:



Given an investment possibility of $325 000 that will immediately produce rental income of $7 500 per month, how long must the investment be held to yield 10% per annum?[199]

Solution:

FIN TVM Begin		payments are due at the begin of each period,
12 per/a	12.00	months per year,
10 i%/a	10.00	% annual interest rate,
325000 PV	325 000.00	investment;
7500 PMT n_{PER}	53.43	months.

Example 2 for finding the number of periods for an annuity due:

If you deposit $50 a month in a savings account that pays 6% interest, how long will it take to reach $1 000?

Solution:

12 per/a	12.00	months per year,
6 i%/a	6.00	% annual interest rate,
0 PV	0.00	start balance,
1000 FV	1 000.00	future value;
50 PMT n_{PER}	19.02	months.

Example for finding the interest rate for an annuity due:

Equipment worth $12 000 is leased for 8 years with monthly payments in advance of $200. The equipment is assumed to have no salvage value at the end of the lease. What yield rate does this represent?

[199] I frankly admit I understand neither this problem nor its solution.

Solution:

12 per/a	12.00	months per year,
8 ENTER↑ 12 × n_{PER}	104.00	payment periods in total,
12000 PV	12 000.00	start value of equipment,
0 FV	0.00	final value of equipment,
200 PMT	200.00	payments;
i%/a	13.07	% annual yield.

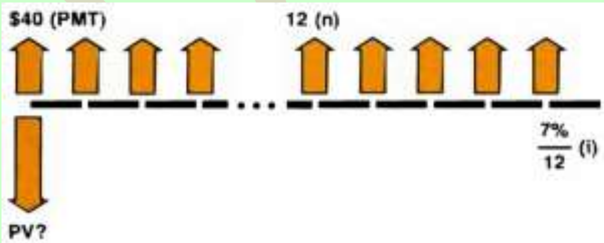
Example for finding the payment amount for an annuity due:

The owner of a building presently worth $70 000 intends to lease it for 20 years at the end of which time he assumes the building will be worthless (i.e., has no residual value). How much must the quarterly payments (in advance) be to achieve a 10% annual yield?

Solution:

4 per/a	4.00	quarters per year,
20 ENTER↑ 4 × n_{PER}	240.00	payment periods in total,
10 i%/a	10.00	% annual target yield,
70000 PV	70 000.00	***PV*** of the building;
0 FV	0.00	***FV*** of the building;
PMT	1982.27	quarterly payments.

Example for finding the present value for an annuity due:

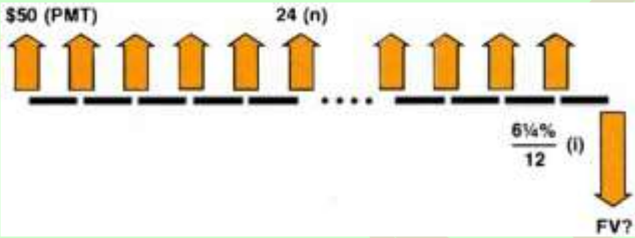


The owner of a down-town parking lot has achieved full occupancy and a 7% annual yield by renting parking spaces for $40 per month payable in advance. Several regular customers want to rent their spaces on an annual basis. What annual rent, also payable in advance, will maintain a 7% annual yield rate?

Solution:

12 per/a	12.00	months per year,
12 n_{PER}	12.00	payment periods in total,
7 i%/a	7.00	% annual target yield,
40 PMT	40.00	monthly payments;
PV	464.98	equivalent annual payment.

Example for finding the future value for an annuity due:



If you can afford to deposit $50 per month in an account with 6 ¼ % interest compounded monthly, how much will you have 2 years from now?

Solution:

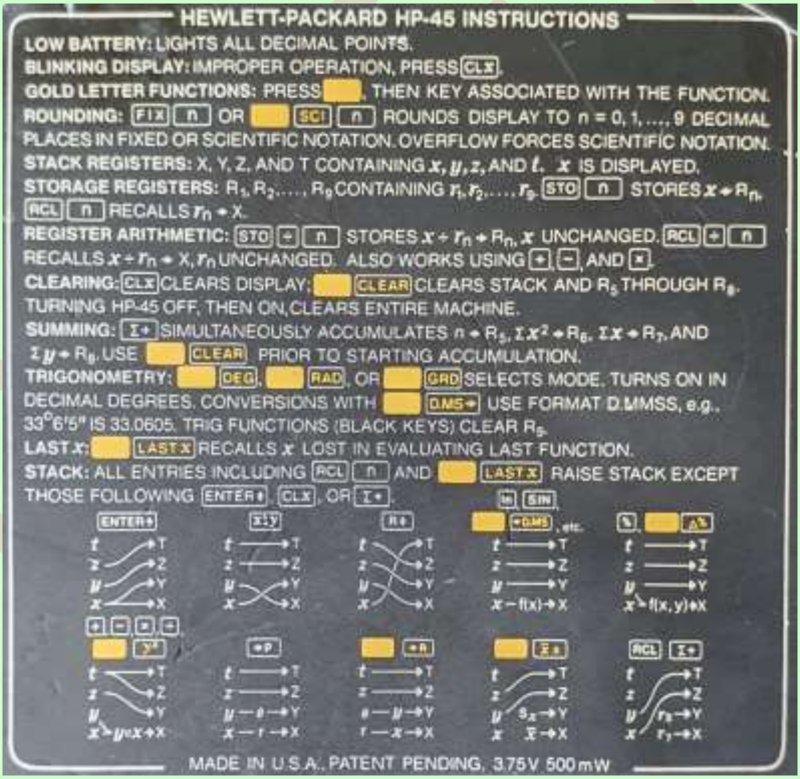
12 per/a	12.00	months per year,
2 ENTER↑ 12 × n_{PER}	24.00	payment periods in total,
6.1.4 i%/a	6.25	% annual target yield,
50 PMT	40.00	monthly payments;
0 PV	0.00	start balance;
FV	1 281.34	balance after two years.

APPENDIX 4: POWER SUPPLY

Your *WP 43S* is powered by a single *CR2032* coin cell (3 V). Alternatively, it may be powered through its USB port – running with even higher speed then. Watch p. 16 and see the *ReM*, *App. A* for more.

WARNING: Removing the battery for longer than xxx *seconds* may erase all data in RAM – only data in *flash memory* will remain.

See what sufficed for explaining the basic functionality of the *HP-45* on its back in 1973:



Though it featured only 59 functions, neither *menus*, *catalogs*, *data types*, browsers, applications, advanced operations (just four statistical sums, means, and standard deviations), named variables, programming, nor customizing – but your *WP 43S* does.

APPENDIX 5: TIME LINE OF QUOTED MANUALS

HP-35 OM	1972
HP-55 OH, HP-21 OH, HP-25 OH	1975
HP-27 OH, HP-67 OHPG	1976
HP-97 OHPG, HP-32 OH, HP-33 OH	1978
HP-34C OHPG	1979
HP-41C/41CV OHPG	1980
HP-16C Computer Scientist OH	1982
HP-15C OH	1987
HP-27S OM, HP-42S OM	1988

In 1976, *Continuous Memory* was a breathtaking innovation; and a grand total of 72 built-in functions, 8 *GP* storage registers, and 49 merged program steps sufficed for professional engineers and scientists doing their work as well as for students striving for their Ph.D. Note that 200 US$ of 1976 correspond to 911 US$ of today! Though linear regressions, correlations, and forecasting had to be programmed by you if you needed them – the respective routine as recommended by *HP* took 44 precious steps.

APPENDIX 6: RELEASE NOTES

	Date	Release notes
0	29.11.12	**Official project start with first publication of the 43S concept and a layout on one of the forums of the *Museum of HP Calculators*** (https://www.hpmuseum.org/cgi-sys/cgiwrap/hpmuseum/archv021.cgi?read=234685#234685)**.** Though there are found far older traces of a '*43S*' denoting a '*Super HP-42S*', though in various more or less fictional cases – pure vapourware™.
0.1	2.2.14 23.5.15	Manual setup based on the one of *WP 34S*. Passed to *Jake Schwartz*, *Eric Smith*, and *Richard Ottosen* for first information.
0.2	3.10.15	Update based on Jake's feedback and further thoughts, distributed to *Eric*, *Jake*, *Marcus*, and *Pauli*.
0.3	21.3.16	Split the manual in three; moved LBL onto the keyboard, renamed STOM to STOCFG, RCLM to RCLCFG, SERR to s_m, and $SERR_w$ to s_{mw}; refined the *Key Response Table*. Passed to *Michael Steinmann* for information.
0.4	28.3.16	Renamed LOGS to EXP and EEX to E. Added hardware information from 2nd manufacturer.
0.5	29.10.16	Returned EEX. Changed keyboard layout.
0.6	22.8.17	Merged the Applications and Owner's Manual. Changed the input order of complex number parts on Pauli's request. Changed keyboard layout introducing D.MS, SST, BST, and % while removing ŷ, RAN#, 'FRC, and 'CFIT. Put 'CFIT into 'STAT and 'FRC into 'MODE. Placed OFF below EXIT for easier customizing. Renamed cc to C5, EEX to E, STOPW to TIMER, SHOW to REGS, 'SOLVE to 'ADV, DLINES to DSTACK, 12h to CLK12, and 24h to CLK24. Replaced IND by →. Deleted %MG since covered by Δ%, added EIGVAL and EIGVEC. Swapped CNST and CONST. Defined the echo rows for alphanumeric and command input. Expanded and modified the character sets for better use of display space. Added the *QRG*.
0.7		Changed keyboard layout. Replaced the labels BST by ≡▲, SST by ≡▼, and UNDO by ↶; added some *alpha input mode* reminders on the keyboard. Added AGRAPH, CLLCD, EQ.xxx, HYP, J/G, M.GOTO, ORTHOF, PIXEL, POINT, TDISP, and 🖶USER. Moved the background considerations out of *ReM App. D*. Introduced **K** as *alpha register* for alphanumeric constants in programs. Removed *fraction data type*. Extended *items* from 6 to 7 characters to match *HP-42S*.

	Date	Release notes
	2.4.18	Specified *data types* more precisely in *ReM App. D*. Reduced the maximum number of *local registers* from 888 to 100. Deleted JG1582 and JG1752. Renamed two commands for TVM. Replaced the heading apostrophe for *menu* names. Put SUMS in STAT. Renamed the trigonometric and hyperbolic functions according to mathematical standards, and 🖶CHR to 🖶CHAR. Redistributed the chapter about constants. Modified STATUS display. Refined the unit conversions to ensure *SI* on one side. Specified 0 SEED. Expanded *ReM App. A*. Added formula output for L.R. Modified CPX?, DBL?, and REAL?. Changed output of binary tests for compatibility with *HP-42S*.
0.8	7.5.18	Changed keyboard layout: introduced TRG containing trigonometric functions, removed HYP into EXP and [π] to g-shifted [/], swapped some shifted labels. Refined the chapters about register arithmetic, *Command Parameter Input*, *Alphanumeric Input*, *Matrix Calculations*, and *Orthogonal Polynomials*. Introduced CLCVAR and more vintage examples. Rearranged *temporary information* on the screen. Renamed REGS to RBR and CLx to CLX. Deleted ANGLE.
	20.9.18	Corrected errors and inconsistencies. Added one more example. Moved the key response table into an appendix.
0.9	3.1.19	Removed *angle data type*. Added another industrial application and many more examples. Exchanged keyboard pictures due to changed bezel. Expanded *App. B*. Added SHOW for displaying full precision of *DP* numbers and FBR for browsing our two fonts. Split a chapter. Expanded some titles. Added the overlay drawing. Modified functionalities of [EXIT] and [1/x] to match *HP-42S*. Added a chapter about curve fitting. Modified functionalities of [ENTER↑] and [↶]. Expanded *App. K.* Renamed DOUBLE to →DP. Added →SP and conversions of *quarts*. Rearranged X.FN. Replaced [USR] by [UM]. Changed keyboard moving [UM], [√x], and [TRI]. Moved 🖶 to [f] [R/S]. Added XIN and XOUT. Added a chapter in *App. E* and information about infinite integers Extended the domain of GCD and LCM. Refined and corrected.
0.10	3.3.19	Returned *angle data type* and αSR. Added IDIVR and VANGLE. Refined FP, IP, IMPFRC, PROFRC, SDIGS?, →DP, →HR, →INT, →REAL, →SP, explanation of ALL, the summary of integer functions, and handling of long alpha strings. Modified contents of CPX, MATX, and α•. Added a summary of matrix functions. Removed the [ON]-key combinations. Modified MEM?. Rewrote the angular conversions. Renamed infinite and finite integers to *long* and *short integers*. Added a chapter about ±∞ and NaN. Modified RBR and the menu for STO and RCL. Removed 🖶 from the keyboard. Renamed X_u to X_e for the distributions. .

	Date	Release notes
0.11	8.5.19	Changed keyboard making [CC] primary and user mode shifted, removing x^2, x⁝, and DSP, adding |x|, DROP, and SHOW, and moving some shifted labels. Modified BITS, CLREGS, CNST, CPX, DISP, EXP, INTS, MODE, PARTS, SHOW, STAT, U→, αMATH, the division matrix, *data type* conversions, and the *Quick Reference Guide*. Added conversions of *barrels*, *carats*, and *fathoms*. Deleted DSP. – Separated predefined variables. Refined *Sect. 6*. Added $\bar{x}_H$, $\bar{x}_{RMS}$, nine statistical sums and five curve fit models. Split STAT in STAT and SUMS; renamed RMDR to RMD, L_n to L_m, $L_{n\alpha}$ to $L_{m\alpha}$, Π to Π_n, Σ to Σ_n, and some constants to avoid search ambiguities. Refined *App. J*, *Sect. 3 and 4*, →INT, CLR, and the functions of [▲] and [▼]. Put SUMS instead of RMD on the keyboard, moved ADV, BITS, CATALOG, EQN, FILL, INTS, MATX, MODE, PROB, RTN, SHOW, STAT, and α.FN. Rearranged A…Ω and *Sect. 2* of the *OM*.
0.12	16.10.19	Rearranged the appendices of the *ReM* from *App. D* on. Expanded *App. A* of the *OM* and *App. K*. Deleted the standardized normal distribution Φ and rearranged PROB. Updated CNST following *CODATA 2018*. Renamed the angular conversions. Changed the composing and cutting functionality of [CC]. Refined exiting *short integer* input. Expanded *App. D*. Specified maximum size of *long integers*. Changed keyboard adding ∡, moving CPX, FIN, RBR, R↑, and SHOW, removing %. Renamed VANGLE to V∡. Modified CPX, MATX, TRI, and X.FN. Rearranged *Section 1* of the *OM*. Added some internal *data types* to *App. B*; reduced the range of *long integer* results and *DP* real inputs to $10^{\pm 999}$. Defined the domains of e^x-1, IDIVR, LN(1+x), MOD, and RMD according to the *HP-42S*; modified PLOT and Σ+. Refined the *Addressing Tables*. Added a *data type* matrix for IDIVR. Refined the *Special Results* in *App. B*.
0.13	30.11.19	Expanded the alpha keyboard and *App. I*. Modified CPX, INTS, MODE, PROB, STK, TEST, α●, SHOW, and STATUS. Refined the sorting order of *items*, ALL, CX→RE, MEM?, RE→CX, RBR, RM, SLVQ, and U→. Started filling *App. F* and *G*. Refined *App. 2*. Added a *long integer* example, CPXR?, LZ?, $\Delta\nu_{Cs}$, conversions of *hectares*, and a proposal for system status information.
0.14	7.3.20	Introduced *system flags* for status information. Split I/O. Added CATALOG'SYS.FL, PRINT, PROG, RANI#, VAR, auxiliary constants, some predefined variables, and an index in *App. I*. Changed keyboard swapping MODE and FLAGS, U→ and ∡→, moving CPX, FILL, RBR, R↑, USER, α.FN, αINTL, √x, and 🖶, displaying PRINT, RMD, STATUS, x^2, and ':', and removing [c/d], 🖶x, →SP, and →DP. Renamed DISP to DSP and SUMS to Σ, changed [↶] to [↺]. Refined the addressing tables and catalog access, [a b/c], ADV, BATT?, BITS, CATALOG'CHARS and 'MENUS, CLALL, CLFALL, CPX, EXP,

	Date	Release notes
		GAP, INTS, I/O, MODE, NEIGHB, PARTS, PRIME?, P.FN, SHOW, STAT, STK, X.FN, αINTL, and α●. Deleted all 16-digit (i.e. *SP*) *data types* as well as A…Z and the commands CLK12, CLK24, CPXi, CPXj, CPXRES, CPXR?, DBL?, DENANY, DENFAC, DENFIX, ENGOVR, FAST, IMPFRC, LZOFF, LZON, LZ?, MULT×, MULT·, POLAR, PROFRC, QUIET, RDX., RDX,, REALRE, RECT, SCIOVR, SLOW, SSIZE4, SSIZE8, →DP, and →SP. Corrected.
0.15	14.6.20	Added BESTF?, RANGE, RANGE?, REGIST, SNAP, and s(a), as well as errors 28 and 31 – 35. Changed DSZ and ISZ to comply with *HP-16C*. Changed keyboard shifting N, O, P, and Q, swapping ? and Z, moving CNST, CPX, FLAGS, RBR, RTN, R↑, VIEW, and ⎙, removing **:**, and adding MOD, ✓, and SNAP. Renamed DSP to DISP, CNST to CONST, CONST to CNST, ASL.BLK to ASLIFT, SSIZE to SSIZE8, TDM to TDM24, and the left and right sided probabilities. Refined ASSIGN, CATALOG, CNST, DISP, INFO, NEXTP, PRIME?, PROB, RBR, RESET, SHOW, SINC, STAT, U→, VIEW, x=+0?, x=−0?, y^x, α→x, ∡, pp. 54 – 57 and 205 – 207 (and consequences) as well as *Section 6* of the *OM*, pp. 108 – 117, *App. B*, *C*, and *E* of the *ReM*, and some looping and statistical explanations. Reduced the maximum number of *local registers* from 100 to 99. Changed ALLSCI to ALLENG and RECTN to POLAR. Added *data type* matrices for powers. Corrected.
0.16	4.7.20	Added torque and mmHg conversions, x_{max} and x_{min}, ISM, and LOADV. Added UNDO to the *IOI*. Refined I/O and the descriptions of LOAD, LOADSS, RESET, and UNDO. Marked the not-undoable *items* in the *IOI*. Renamed the constants according to the *OM* and kicked them out of the *IOI*. Corrected.

INDEX

This index lists special terms and keywords used in this manual. Furthermore, it points to the most prominent of the 167 examples included, marked '(**ex**.)'.

Items are listed below only if they are extensively treated in this manual (remember you find each and every *item* provided explained in the *IOI* printed in the *ReM*; and the *IOI* will also point you to further explanations if applicable). Looking at the *Table of Contents* above is recommended as well – titles are not repeated below.

42 (ex.) 134
about to die (ex.) 194
account balancing (ex.) 31
address space 53
addressing, indirect 64
ADM 124
advertising pictures (ex.) 95
AIM 192
air pressure and altitude (ex.) 86
aircraft navigation (ex.) 129
alpha input mode 76, 192
altimeter (ex.) 86
angular display mode 124
APR 310
archery statistics (ex.) 99
arithmetic shift 141
ASSIGN 280
automatic stack lift 35
balloon trip (ex.) 84
base conversions 138
bicycle gearing (ex.) 85
bit numbering 139
black or white cats (ex.) 195
Boolean operations 141
branching 213, 217
bubble sort (ex.) 218
caesium-137 91
calculator stand (ex.) 152
cannery (ex.) 204
carbon-14 dating (ex.) 240
carry 140
CDF 97
chain calculation (ex.) 34
chi-square statistic (ex.) 106
cleaning a veranda (ex.) 277
closing numeric input 24
complex aircraft navigation (ex.) 157
compound interest (ex.) 48
confidence limits (ex.) 108
connecting peaks (ex.) 94
continuous distribution 97
cross product (ex.) 159, 176
cubic inches (ex.) 277
current step 203
data type 68
data type conversions 74
dating (ex.) 189
debugging 212
decrement and skip 217
dice from *Las Vegas* (ex.) 107
diffraction pattern (ex.) 257
digit group separation 79
discrete distribution 97

dot product 158, 176
dyadic functions 31
earthquakes (ex.) 89
electron-volts (ex.) 273
engineer's notation 80
enter exponent 24
ENTER↑ 33
error probability 97
extrapolation (ex.) 105
falling around the earth (ex.) 226
falling in *Pisa* (ex.) 102
falling with drag (ex.) 224
fencing land (ex.) 19
FILL 38
filling tires (ex.) 277
fixed point notation 81
flags 54, 55
forecasting (ex.) 105
free fall (ex.) 102
Fukushima accident (ex.) 91
general purpose registers 44, 54, 55
Golden Bow (ex.) 99
great circle distance (ex.) 126
Greek letters 192
Horner scheme (ex.) 49
improper fraction 151
increment and skip 217
indicators, status bar 75
indirect addressing 61, 62
integer sign mode 136
interpolation (ex.) 104
ISM 136
item 26
keystroke programming 201
logical operations 141
long integers 134
Mach number (ex.) 44
markup and margin (ex.) 121
measuring capability (ex.) 114
measuring system analysis (ex.) 112
menu 26
menu section 27
menu view 26
modulo 143
monadic functions 30
Mother's Kitchen (ex.) 204
Mt. Everest (ex.) 87
MyMenu 285
Myα 290
navigating in space (ex.) 131
overflow 137
parachutist (ex.) 224
PDF 97
PEM 21, 202, 204, 245, 262
PMF 97
prefix 18
primary function 17
process capability 100
program editing 210
program pointer 203
proper fraction 151
proton in magnetic field (ex.) 176
quantile function 98
R↓, R↑ 37
radioactivity (ex.) 91, 240
radioactivity (units) 279
radix mark setting 79
RCL 58, 61
remainder of division 142
Rigel Centaurus (ex.) 50
rotate bits 140
rotate text 197
RPN 14, 23, 31, 37, 40, 42, 45
satellite orbits (ex.) 226
scientific notation 80
scrap rate (ex.) 108
search text 197
secondary function 17
shift bits 140

shift text 198
short integers 135
significant change (ex.) 114, 116
significant improvement (ex.) 101
Sirius (ex.) 50
skiing (ex.) 83
skydiving (ex.) 224
softkey 27, 28
sorting numbers (ex.) 218
special registers 54
squaring circles (ex.) 82
stack 32
stack overflow 43
startup default 52
statistical registers 52, 54
status bar 75
submenu 27
surfaces of *Jupiter's* moons (ex.) 21
TAM 56, 209, 260
tax deduction (ex.) 122
temporary alpha mode 56
temporary information 68
Tower of Pisa (ex.) 102
triadic functions 36
Upper Lagunia (ex.) 94
user flags 55
variables 56
vector operations in 2D 177
VIEW 59
virtual keyboard 57, 193, 209
Willie's Widget Works (ex.) 95
XEQ 22