USER MANUAL

Revision 4a
Nov-11-2022

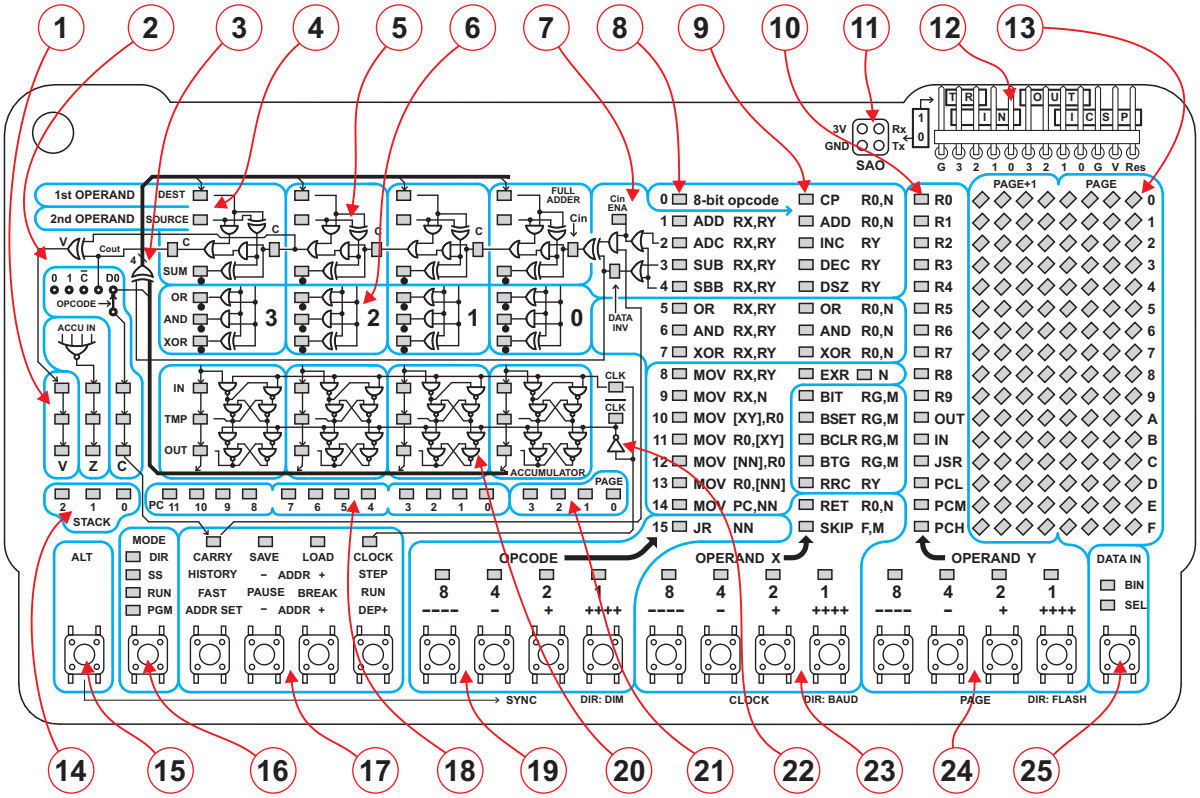
Button Commands

MODE	COMMAND KEYS				OPCODE	OPER X	OPER Y
DIR	**Carry**	**Save**	**Load**	**Clock**	**Opcode**	**Oper X**	**Oper Y**
	Toggle Carry Flag	Send Program Memory to Serial Port	Load Program Memory from Serial Port	Master Clock source	Instruction Opcode (bits 11-8)	Direct Operand X or Opcode (bits 7-4)	Direct Operand Y (bits 3-0)
ALT		Save Program Memory to selected Flash	Load Program Memory from selected Flash		Dimmer level select	Baud Rate select	Flash portion select for Save / Load
SS	**Historry**	**- Addr**	**Addr +**	**Step**	**Opcode**	**Oper X**	**Oper Y**
	Enter History submode	Decrement Program Memory Address	Increment Program Memory Address	Execute one instruction	Instruction Opcode (bits 11-8)	Direct Operand X or Opcode (bits 7-4)	Direct Operand Y (bits 3-0)
ALT	Toggle Carry Flag	Reset Program Memory Address to 0x000	Preset Program Memory Address to the last word used	Address set from Opcode, Operand X and Operand Y	User Sync select	Processor Clock select	Display Page select
RUN	**Fast**	**Pause**	**Break**	**Run**	**Opcode**	**Oper X**	**Oper Y**
	On/Off Toggle 10× Faster Clock and Sync	Program Execution Pause / Resume	Terminate Program Execution	RUN Program From Program Memory	—	—	—
ALT					User Sync select	Processor Clock select	Display Page select
PGM	**AddrSet**	**- Addr**	**Addr +**	**Dep +**	**Opcode**	**Oper X**	**Oper Y**
	Address set from Opcode, Operand X, Operand Y	Decrement Program Memory Address	Increment Program Memory Address	Write 12-bit word to Program Memory and inc PC	Instruction Opcode (bits 11-8)	Direct Operand X or Opcode (bits 7-4)	Direct Operand Y (bits 3-0)
ALT	Delete the current word (move all subsequent words down)	Reset Page and Pgm Memory Address to 0x000	Preset Program Mem Address to the last word used	Duplicate the current word (move all subsequent words up)			
	ALT+Both Keys → Pressed	Clear All Memory					

Gray = **ALT** key pressed

Indicators, buttons and connectors

1. **STATUS** register, with **V** (**oVerflow**), **Z** (**Zero**) and **C** (**Carry**) flags
2. **Flag Logic**, which generates signals for flags
3. **Data Buffer/Inverter**, switched by Carry Logic **(7)** and used for Adder/Subtractor
4. **Operands**, represented and indicated as inputs to **ALU** unit
5. **Arithmetic** unit (4-bit **Full Adder / Subtractor**) as a part of **ALU** unit
6. **Logic** unit (4-bit **OR / AND / XOR** gates) as a part of **ALU** unit
7. **Carry Input Logic** (used for **Data** and **Carry Inversion** in case of subtraction)
8. **Opcode Decoder** output (also used as interactive **Code Disassembler**)
9. **Operand X Decoder** output (also used as interactive **Code Disassembler**)
10. **Operand Y Decoder** output (also used as interactive **Code Disassembler**)
11. **SAO** (**"Shitty Add-On"**) connector, with **Ground**, **+3V** and **UART Rx/Tx** pins
12. **I/O** connector, with **Input** and **Output** ports and **PIC MCU** Programming pins
13. **LED Matrix** which displays two pages (**2×16** nibbles) of **Data Memory**



14. Three-bit **Stack Pointer** indicator
15. **ALT** button, which switches some indicators and buttons to alternate functions
16. **MODE** button, used to switch between **Direct/Single Step/Run/Program** mode
17. **Command Group** of buttons, with Mode-specific fuctions
18. **Program Memory Address** pointer (a.k.a. **Program Counter**, **PC**)
19. **Opcode** buttons and indicators (instruction bits **11-8**)
20. Symbolic representation and indicators for **Accumulator** register
21. **Page** register, determines which page of **Data Memory** is shown on LED matrix
22. **Master Clock** signal inverter, used for **Master-Slave Flip-Flops** triggering
23. **Operand X** buttons and indicators or **Opcode** extension (instruction bits **7-4**)
24. **Operand Y** buttons and indicators (instruction bits **3-0**)
25. **Data In Select** button and indicator, switches between **Binary** and **Select** mode

Indicators, buttons and connectors

On-Off Button

The only button available at the bottom side is the **On-Off** button. Switching **On-Off** is possible in every mode, and also when the user's program is running. In the **Off** state, clock signal is halted, the system processor is in Sleep mode and all outputs on the **I/O** connector (12) are in the high-impedance state. However, there are **pull-up** resistors on all inputs and **pull-down** resistors on all outputs. The only exception is the serial **Tx** output, which does not have not **pull-down**, but **pull-up** resistor, as the default level of **Tx** is high. The resistance of every pull-up and pull-down is **22KΩ**.

Switching the unit **Off** does not affect processor's registers or contents of memory or program state, so when the unit is turned **On** again, it will continue the execution as if it wasn't stopped at all.

ALT Button (15)

This is the only button that does not initiate the command execution when it is pressed, but it modifies the functions of other buttons and some indicators. It should be used similarly to the **Alt** button on the computer's keyboard, which means that it should be pressed prior to the button which function should be modified. The main Alt-functions of other buttons is printed under the button bar, and it is depended on the mode selected. Here is the function of buttons in the **Opcode**, **Operand X** and **Operand Y** fields, with the original and modified function in different modes:

	OPCODE buttons and indicator bar		**OPERAND X** buttons and indicator bar		**OPERAND Y** buttons and indicator bar	
MODE	Original	ALT pressed	Original	ALT pressed	Original	ALT pressed
DIR	Opcode	Dimmer	Operand X	Baud rate	Operand Y	Flash Addr
SS	Opcode	Sync	Operand X	Clock	Operand Y	Page
RUN	—	Sync	—	Clock	—	Page
PGM	Opcode	Sync	Operand X	Clock	Operand Y	Page

DATA IN Button and Indicators (25)

Used for **Data Input Method** selection, which affects button groups **Opcode**, **Operand X** and **Operand Y**. The same Data Input Method selection is valid for ALT-functions of the three groups (**Sync, Clock, Page, Dimmer, Baud Rate** and **Flash Address**). Every press of the **Data In** button toggles between the two methods, and the current method is displayed on the **BIN/SEL** indicators.

Here is the description of the two available methods:

1. BINARY Method

In the **Binary** method, every buttons simply toggles the corresponding bit state of the 4-bit selection register, displayed by four indicator above the buttons (Most Significant Bit, marked by "8", is at the left). At the same time, the 16-step indicator bar displays the decoded binary state of the selection register.

2. SELECT Method

Buttons **"-"** and **"+"** are used to decrement and increment by one the 4-bit binary state of the 4-bit selection register. Buttons **"----"** and **"++++"** can be used to decrement and increment the state of the same register by four, which can be used to speed up the selection.

Indicators, buttons and connectors

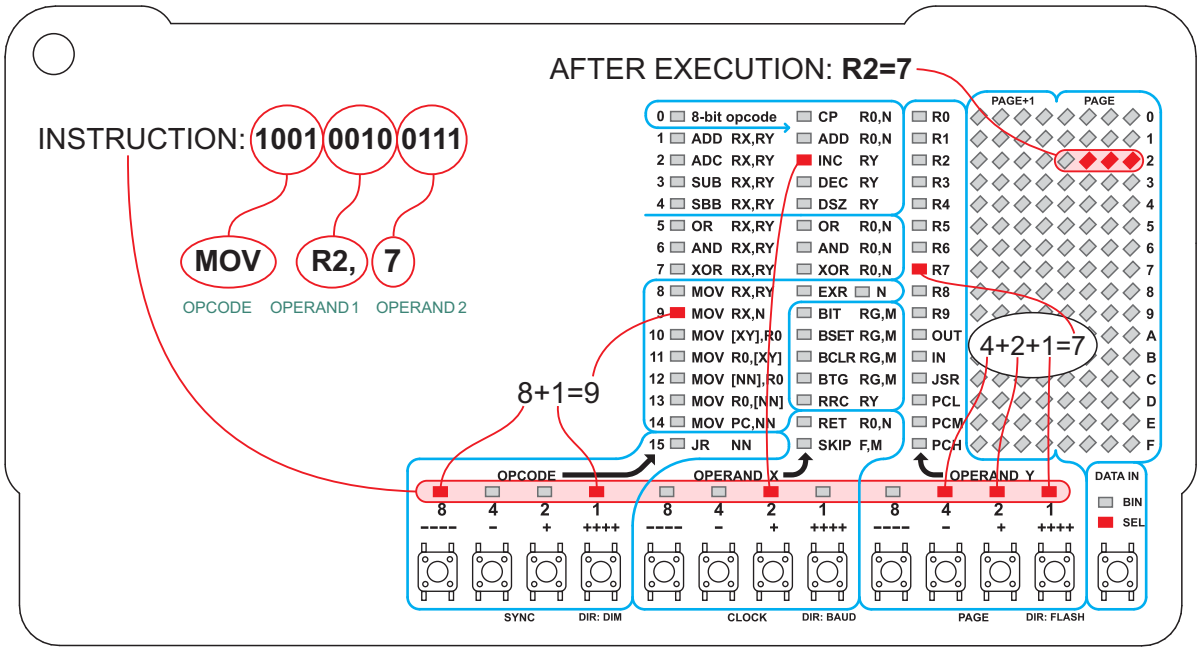
Preface: Bit coding

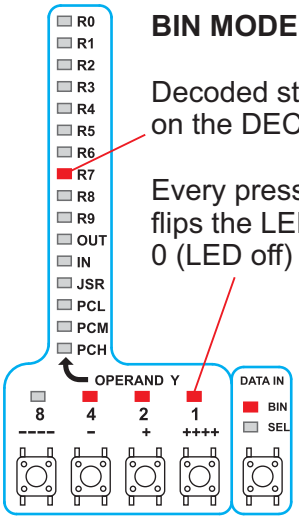
Before we start with the **Opcode** and **Operand** fields, just a few words about the bit coding fields inside the instruction word. As the unit is supposed to be programmed directly in a **machine language** (**binary code**, ones and zeros), special care was taken to simplify the structure of the instruction codes and pack the bit fields in 4-bit groups which are easy to perceive and remember.

The main code fields are **Opcode** and **Operands**. The typical instruction contains one opcode and a flexible number of operands.

Opcode (abbreviated from **op**eration **code**) is the portion of a machine language instruction that specifies the operation to be performed. In this case, the opcode is always located at the most significant bits of the 12-bit instruction word: bits 11-8 or bits 11-4.

Operands are values assigned to registers or memory and they are specified and accessed using more or less complex addressing modes. Here we have instructions which contain **one** or **two** operands (but note that there are processors which support more or zero operands). Generally, **Operands** contain **data**, and **Opcode** tells the processor **what to do** with the data.

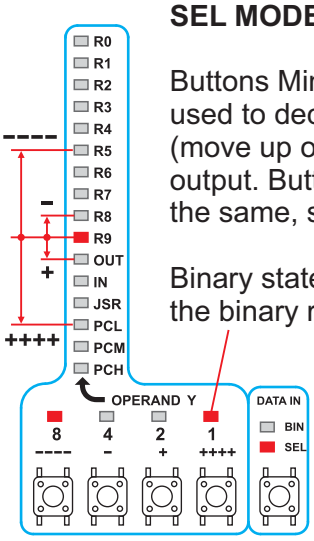




BIN MODE:

Decoded state is displayed on the DECODER column

Every press on the button flips the LED state:
0 (LED off) or 1 (LED on)



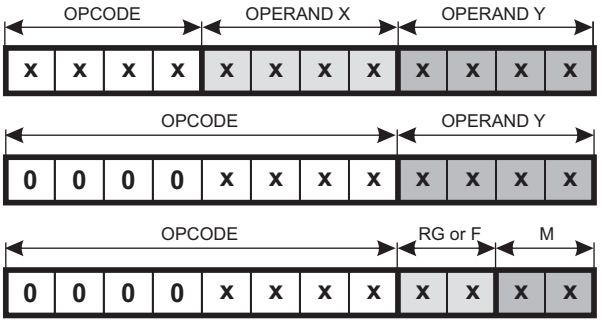
SEL MODE:

Buttons Minus (-) and Plus (+) are used to decrement or increment (move up or down) the decoded output. Buttons (----) and (++++) do the same, skipping 4 values at once

Binary state is displayed on the binary row

Indicators, buttons and connectors

Let's see how the instruction is organized internally. In this processor, every instruction has the same length, which is **12 bits**. Opcode and operands may be located in any part of the instruction code, but in our case, for clarity and ease of programming, the opcode always contains four or eight bits and it is located in the leftmost part of the 12-bit instruction word (highest order bits), and operand or operands are in the rightmost eight or four bits:



x = bit 0 or 1

If the opcode is eight bits wide, then the first four bits are always **0000**. This way of coding gives enough space for a total of **15** instructions with **4-bit** opcode (**0001**-**1111**), and **16** instructions with **8-bit** opcode (**00000000**-**00001111**).

Please note that there are several instructions (**Bit Test/Set/Clear/Toggle** and **Skip**) with the **Operand Y** field split in two 2-bit operands.

OPCODE Buttons and Indicators (19) and decoder (8)

Buttons in the field which is conditionally named **Opcode** are generally used to preset the four upper (most significant) bits of the 12-bit instruction word. The word in binary form is readable on the indicators in the **Opcode (19)** field. The indicator column **(8)** represents the decoded 4-bit nibble from the Opcode field, and it has only one LED in ON state. This column can be conveniently used as the disassembled opcode with the instruction printed next to the LED.

If the **DATA IN** indicator is in the **BIN** state, Opcode can be entered bit-by-bit, and if it is in the **SEL** state, the same buttons are used to move the decoded output up/down by one (buttons **"-"** and **"+"**) or four places (buttons **"----"** and **"++++"**).

In **Single Step** and **RUN** modes, every modification of the **Program Memory Address (PC)** will automatically read the contents of the instruction and update the **Opcode, Operand X** and **Operand Y** indicators. You can modify it using buttons, end even execute the new state in **SS** mode (by pressing button **STEP**), but it will not affect the contents of the Program Memory. The only way to alter the contents of the Program Memory is to press the **DEPosit** button in **PGM** mode.

Indicators, buttons and connectors

OPERAND X Buttons and Indicators (23) and decoder (9)

Buttons and indicators in the **OPERAND X** group are used to preset the central nibble (bits 7-4) of the instruction. If the **Opcode** field contains bits 0000, **Operand X** field does not represent the operand anymore, but the extension of the **Opcode** field.

The word in binary form is readable on the indicators in the **Operand X (23)** field. The indicator column **(9)** represents the decoded 4-bit nibble from the **Operand X** field, and it has only one LED in ON state. This column can be conveniently used as the disassembled opcode, but the special care should be taken if the **Opcode** field contains **0000**, as the instruction printed next to the LED is valid only in that case. In all other cases, the decoded output should be treated as the register name printed next to the LED matrix field (**Data Memory**).

If the **DATA IN** indicator is at the **BIN** state, **Operand X** can be entered bit-by-bit, and if it is in the **SEL** state, the same buttons are used to move the decoded output up/down by one (buttons **"-"** and **"+"**) or four places (buttons **"----"** and **"++++"**).

In **Single Step** and **PGM** modes, every modification of the **Program Memory Address (PC)** will automatically read the contents of the instruction and update the **Opcode, Operand X** and **Operand Y** indicators. You can modify it using buttons, end even execute the new state in **SS** mode (by pressing button **STEP**), but it will not affect the contents of the Program Memory unless you press the **DEPosit** button in **PGM** mode.

Column **(9)** contains one extra indicator, which is in the instruction **EXR R0,N** field. This instruction exchanges a group of **General Purpose** and **Special Purpose** registers from the **Page 0** with the equivalent number of nibbles in the **Page 14** of Data Memory, and the indicator is flipped every time when the instruction is executed. So the indicator should be **ON** only when register contents are exchanged between **Page 0** and **Page 14** and **OFF** when they are flipped back to their original positions. That couls help keeping track of program execution.

In **Single Step** and **RUN** modes, the function of **Operand X** buttons and indicators is modified when the **ALT** button is depressed. In that case, **Clock** register is accessible instead of **Operand X** register. **Clock** register is used to adjust the processor speed.

OPERAND Y Buttons and Indicators (24) and decoder (10)

Buttons and indicators in the **OPERAND Y** group are used to preset the lower nibble (bits 3-0) of the instruction. The word in binary form is readable on the indicators in the **Operand Y (24)** field. The indicator column **(10)** represents the decoded 4-bit nibble from the **Operand Y** field, and it has only one LED in ON state. This column can be conveniently used as the disassembled opcode, and it is valid for most instructions, but not for instructions **Bit Test/Set/Clear/Toggle** and **Skip**, which split the field which we named as **Operand Y** in two 2-bit operands. For these instructions, decoding should be performed manually.

If the **DATA IN** indicator is at the **BIN** state, **Operand Y** can be entered bit-by-bit, and if it is in the **SEL** state, the same buttons are used to move the decoded output up/down by one (buttons **"-"** and **"+"**) or four places (buttons **"----"** and **"++++"**).

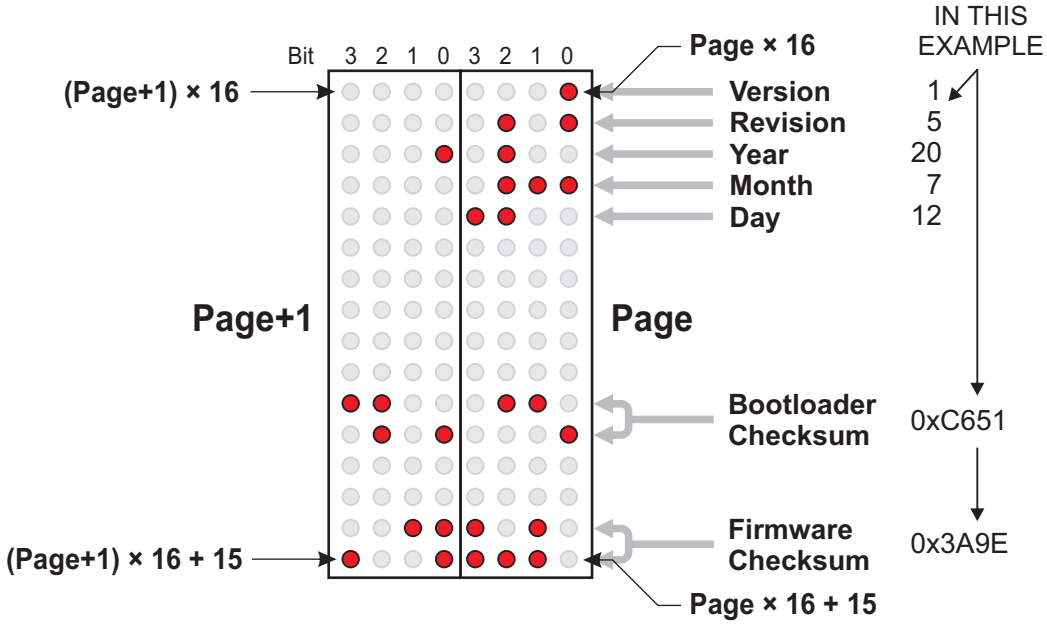
In **Single Step** and **PGM** modes, every modification of the **Program Memory Address (PC)** will automatically cause reading of the contents of the new instruction and update the **Opcode, Operand X** and **Operand Y** indicators. You can modify it using buttons, end even execute the new state in **SS** mode (by pressing button **STEP**), but it will not affect the contents of the Program Memory unless you press the **DEPosit** button in **PGM** mode.

In **Single Step** and **RUN** modes, the function of **Operand Y** buttons and indicators is modified when the **ALT** button is depressed. In that case, **Page** register is accessible instead of **Operand Y** register.

Indicators, buttons and connectors

LED Matrix (Data Memory) (13) and Page indicator (21)

Data Memory display is visually organized as **16×8** matrix, but it is functionally divided in two **16×4** displays. The right **16×4** half displays the contents of one Data Memory page defined by the state of the **Page** register (21), and the left half is for the next one (**Page+1**). The whole Data Memory contains **256 nibbles**, which gives a total of **16** pages, and if the right half displays the last page (**Page 15**), then the left half is wrapped to the beginning of the address space as **Page 0**. This enables watching both **General Function Registers** (on **Page 0**) and **Special Function Registers** (**Page 15**) at the same time.



Data Memory display (13) is disabled in **DIRect** and **PGM** modes, but in **DIRect** mode it has the special function when the **ALT** button is pressed. Then it displays the occupancy of **16 Flash Memory blocks**, which can help in **Flash Memory** organization and navigation.

Also, after the **Master Reset,** Data Memory Display in **DIRect** mode (which is default after **Reset**) shows the **Version/Revision/Year/Month/Day** numbers or the firmware release at the first five rows of the **LED** Matrix. In the middle of the matrix (rows **10** and **11**) there is the **Checksum** of Program Memory for the **Bootloader Segment**, and, on the two bottom rows, the **Checksum** for the **General Segment** (main firmware).

Master Reset is possible only after the batteries are disconnected and then reconnected, or when pins **G** (Ground) and **Res** (Reset) of the **I/O Connector** are shortened externally. After any button is pressed, this data is cleared from the display.

Note that **Master Reset** also clears all **Program** and **Data Memory**, but not the contents saved in the internal Flash.

Data Memory display can be disabled under the program control, if bit **2** (**MatrixOff**) in the register **WrFlags** (Address **0xF3**) is set. If bit 3 (**LedsOff**) in the register **WrFlags** (Address **0xF3**) is set, all other **LEDs** will be disabled, only the **LED CLK** or **INVERSE CLK** (on the schematic drawing) will still be **ON**. These **LEDs** are the indicator that the unit is in operation (or, if they are alternatively blinking, that it is running), and they can not be disabled.

Page is the 4-bit register which is accessible to the user's program in the **Special Function Register** (**SFR**) group, in the data memory address **0xF0**. In modes **SS** and **RUN** it can be easily preset manually, when the **ALT** button is depressed and the **Operand Y** buttons are used for Page contents adjusting.

At every program **Run**, the **Page** register is reset to **0000**. Mode **Single Step** (**SS**) has its own **Page** register, so if some value was preset in **SS** mode, it will be kept and restored at every reentry to the **SS** mode.

Indicators, buttons and connectors

1st and 2nd OPERAND (Input to ALU unit) (4)

This is the first field of the **ALU/Accumulator** data flow, which is in the core of the processor. The data indicated in this field actually do not exist as registers, but only displays the input states to the ALU, and thus makes it easier to follow the process.

Please note the difference between these pairs of terms:
"**Operand X**" - "**Operand Y**",
"**1st Operand**" - "**2nd Operand**", and
"**Source**" - "**Destination**"

In many cases these pairs of terms will mean the same, but there are also cases when there is the difference. Operands **X** and **Y** are simply operands which are defined in **X** and **Y** fields on the panel, and that's all. At the other hand, **1st** and **2nd** operands are just defined by the order of appearance, and **Source** and **Destination** are exactly what these words mean.

There are different rules for different processors, but here the most popular rule is applied. If there are two operands, then the first one is always the destination (sometimes it's also the source), and the second one is always the source. So if the instruction is:

ADD RX, RY

that means "Add Arithmetically contents of **Register Y** to the contents of **Register X** and write the result in **Register X**".

Some instructions have only one operand. In general case, it is source and destination at the same time. For instance:

INC RY

means "Increment the value of **Register Y** by one and write the result in **Register Y**". It's obvious that there was an invisible source, which is the literal "1", added to the **Register Y**.

Sometimes the destination is hidden as the operand, and you have to know the operation defined by the Opcode, to know where the result is stored:

BIT R2, 3

This operation tests bit **3** in register **R2**, so it's the single bit source, but where is the destination? In this instruction, it is the single bit destination, **Flag Z**. To make it more complicated, if bit **3** in register **R2** is **0**, the resulting flag **Z** will be **1**, and vice versa. But it makes more sense when we know that **Z (Zero) Flag** is set **(1)** when the result of the operation is **Zero (0)**.

It is clear that it is not easy to define the operands precisely, so the representation of input signal to the **ALU** will sometimes be inaccurate. Also, captions **"DEST"** and **"SOURCE"** next to the indicators are only roughly informative.

4-bit Full Adder / Subtractor (5)

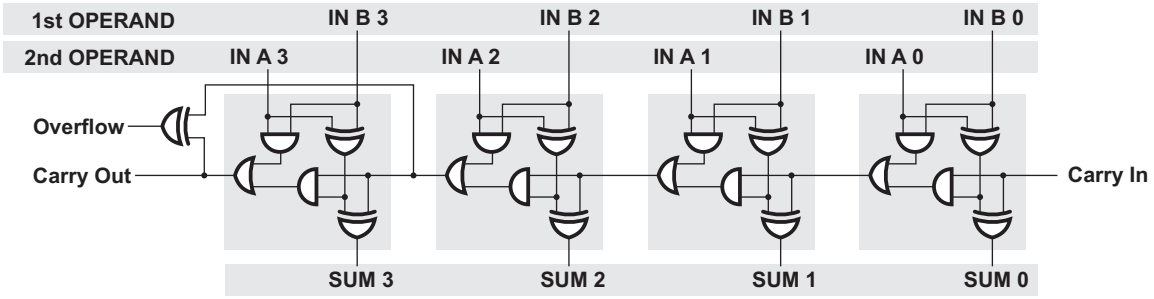
Arithmetic unit performs adding and subtracting operations either with **unsigned positive**, or **Two's complement signed** (either positive or negative) binary numbers. **"Full"** Adder means that it not only adds bits, but also processes **Carry (C)** bit. So every bit stage has three inputs (**A**, **B** and **Carry** from the previous stage) and two outputs (**Sum** and **Carry** to the next stage). The Carry input of the **LSB** (Least Significant Bit) is the **global Carry input** to the adder, and the output from the **MSB** (Most Significant Bit) is the **global Carry output**.

Subtracting is actually adding of negative value, so the source operand is **inverted**. To make it negative in **2's complement** form, it should be also incremented by one, but the inverse **Carry** logic (which is named **Borrow**) in subtraction process compensates this and always gives the correct result, even without adding. If there is the **Borrow** condition (**no Carry**), then the resulting **-1** difference (caused by non-adding **1** at negation) automatically adds **Borrow** (**-1**) to the result, and **No Borrow** (**Carry set**) adds **1** and again compensates **2's complement** negation.

Adder/Subtractor is used not only for **Add** and **Subtract** instructions (with or without **Carry** or **Borrow**), but also for **CP (Compare)** instruction. Compare is actually same as **Subtract**, but the result is not written anywhere but lost, only the flags are preserved.

Indicators, buttons and connectors

It's amazing to learn about the theory of operation of the binary adder/subtractor. Two's complement binary math sometimes looks like magic, when everything turns simple with inverting and negating binary numbers and processing them always in the same adder/subtractor hardware. The Carry logic not only "works" for adding and subtracting, but also for **signed numbers** processing in the same hardware.



In a few words, signed numbers in the **2's complement** math are represented so that **MSB** (leftmost bit) is the **sign** (**"0"** for **"+", "1"** for **"-"**), and all other bits follow the 2'complement rule (inverted bits plus **1**). So there is one bit less for the number representation, as the range for the 4-bit signed number is **-8** to **+7** (for the **8-bit** number, it would be **-128** to **+127**), but everything else is quite simple: adding and subtracting can be performed in the same adder/subtractor hardware!

Adding and subtracting of longer (unsigned positive or signed negative or positive) numbers is performed in serial manner, using **Carry/Borrow** bit for linking. This is the example how to add or subtract two **16-bit** numbers, wether they are unsigned positive or signed negative or positive. If one number is in registers **R0, R1, R2** and **R3**, and the another one in **R4, R5, R6** and **R7**, after this addition (or subtraction) the result will be in **R0, R1, R2** and **R3**:

Addition:	**ADD R0, R4**	Subtraction:	**SUB R0, R4**
	ADC R1, R5		**SBB R1, R5**
	ADC R2, R6		**SBB R2, R6**
	ADC R3, R7		**SBB R3, R7**

Note that the first operation is without **Carry** (or **Borrow**), and all others are with **Carry** (or **Borrow**). This method can be used to add or subtract numbers of any length.

The example uses the **Little Endian** notation (**Least Significant** nibbles are written in the lower address of memory or register file). The principle is the same for **Big Endian** notation (**Most Significant** nibbles on low addresses, **Least Significant** on top), but the order of registers would be reversed (**ADD R3, R7**, then **ADC R2, R6**, and so on). Lowest bits are always processed first.

The same technique is applicable for adding or subtracting of **signed numbers** of any length. The rule is that there is only one Sign bit for the number of any length, always at the **MSB** location of the Most Significant nibble.

For the **unsigned binary numbers**, the global **Carry** (or **Borrow**) bit for the result is available after the last **ADC** (or **SBB**) instruction. If the **Carry Flag** is **set**, that means that the **addition has overflowed** and the result can't fit the register width. For **subtraction**, the outcome is **reversed**: **No Carry** (which means **Borrow Set**) means that the result **has overflowed** (the correct word here is **Underflowed**) and thus not usable.

This was valid for **unsigned** numbers, but for **signed** numbers **Carry Flag** outcome has no meaning, but the **V** (**Overflow**) **Flag** is used instead. So if **V Flag** is **set**, the result has overflowed or underflowed (can not be represented in the existing register) and it is not usable.

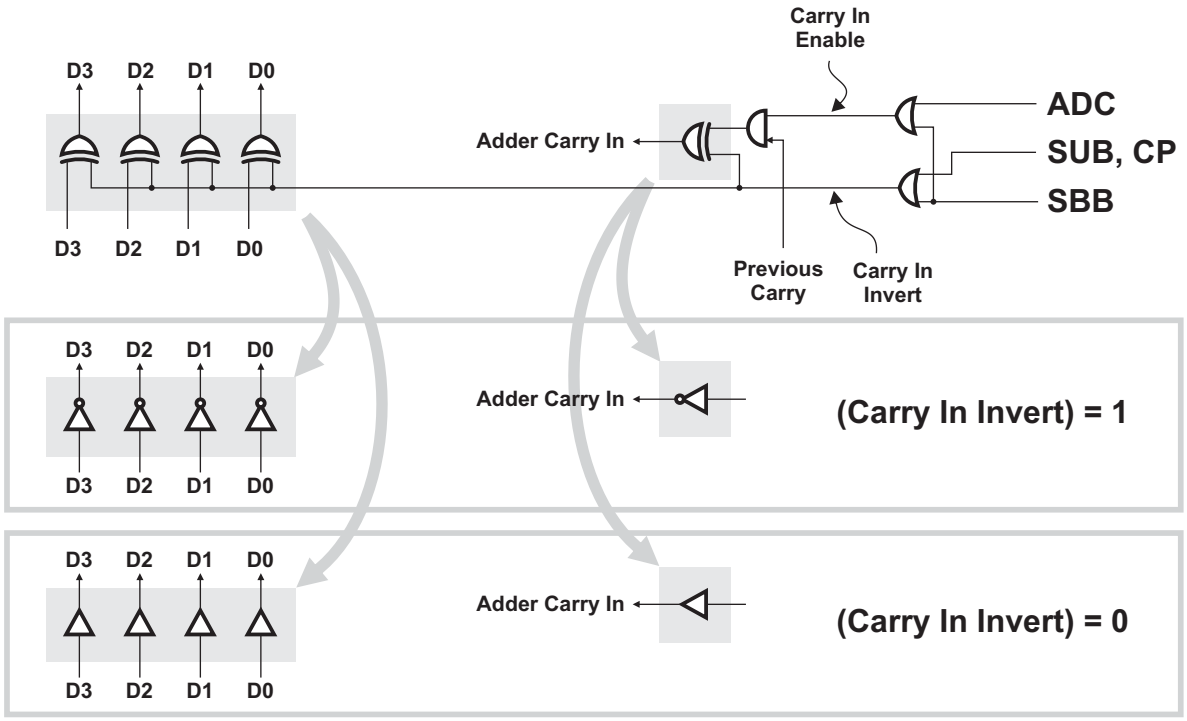
V Flag has no meaning for the **unsigned numbers**, so it is not frequently used. That's why it is not present as a condition in the **SKIP** instruction, but it is available and can be tested (using the instruction **BIT RG, M**) in the **SFR** (**Special Function Register**) named **RdFlags**, bit **1**. The Special Function Register **RdFlags** is at the **Data Memory** address **0xF3**.

Overflow Flag (**V**) is generated in the adder/subtractor hardware circuit in a very simple way: if the **Carry Input** to the last Adder bit and the **Carry Output** from the same adder bit are different, the **Overflow Flag** is set, and that's all. So the single **XOR** circuit does the whole task, and it is just another example of the simplicity and beauty of the adder/subtractor circuit.

Indicators, buttons and connectors

Carry Input Logic (7) and Data Buffer/Inverter (3)

This is the simple logic circuit which inverts **Carry** logic level and **Data Bus signals** for subtraction, and leaves them unchanged (**true logic**) for addition. Also, it switches off the **Carry** input signal (forces it to **Low** for addition, or **High** for subtraction) in the operations which do not process Carry input signal (**ADD, SUB** and **CP**).



Logic **XOR** circuit, which is at the input of the **Adder** and which generates **Cin** signal, simply inverts the **Carry** signal if the **Carry In Invert =1**, and serves as a single buffer (which does not modify the signal level) if **Carry In Invert =0**. The same is valid for all **DATA** signals in the internal **DATA BUS** (there is only one representation of the Data Bus **XOR** circuit on the panel schematics). So both **Carry** and **Data** are inverted if the instruction involves **Subtraction**.

Logic unit (4-bit OR / AND / XOR gates) (6)

This is the second part of the **ALU** unit, where logic instructions are performed. Its structure is quite straightforward, but several facts should be noted before we finish the description of **ALU**.

Due to the lack of space on the panel, the schematic of **ALU** circuit is simplified. One of the circuits that is missing is the complex part of the instruction decoder, which selects not only the result from the adder or some of logic outputs (**OR, AND** or **XOR**), but also the data path from **Data Memory** to the Accumulator inputs, the **I/O** data path, **SFR** logic and so on. The vast majority of the circuits are not represented, simply because it would, even in the simplest possible project, require the panel to be the size of the average table surface, with thousands of gates and many indicators.

Also, the **ALU** circuit is not optimized, but drawn so that it is clear and straightforward. The real **ALU** circuit in the microprocessor looks much less familiar and hard to follow and understand at the first sight, as the number of gates is minimized.

One more thing which is different in modern processors is the **Carry Generator** logic. This “serial” approach works correctly, but it slows down the operation of the processor, due to the propagation delays on the long path, through many gates from the **Carry Input** to the **Carry Output**. Note that there is also (but not represented here) the **Fast Carry Generator**, which works in parallel mode and thus requires more gates, but has much lower propagation delay.

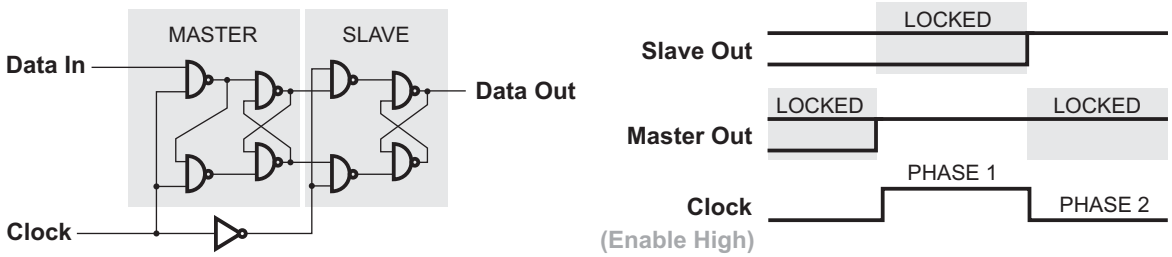
Indicators, buttons and connectors

Accumulator (20) and **Master Clock Signal Inverter** (22)

The first thing that should be noted is that there is not one, but **16 Accumulators** in this processor, and they are named as **Registers R0-R15**. So it's good to imagine them as **16 layers** of the accumulator schematics, and only one is selected and visible, depended which register is the destination one.

Please note that some instructions need no accumulator, as the destination may be the **Data Memory** nibble, **Program Counter**, or even a single bit (**Flag**) in the **Status Register**.

The accumulator contains a series of four Flip-Flops, not a simple ones, but **Master-Slave Edge-Triggered D Flip-Flops**.



In the first phase of the **Clock** signal on the **Master-Slave Edge-Triggered D Flip-Flop** schematics, when the **Clock** signal is **High** (in **Single Step** or **Direct** mode, it's when the button **Step** or **Clock** is pressed), the first **Flip-Flop** is in **transparent** state (unlocked). When the **Clock** logic level is changed to **Low**, the first Flip-Flop (the **Master** one) latches the **Data** logic level and the second (**Slave**) Flip-Flop is in **transparent** state. This two-fold latching solves the problem of circular self-triggering when the same register serves as the **source** and the **destination** at the same time, as one of Flip-Flops is always latched. The **Master Flip-Flop output** can change its state only when the **Clock** signal is **high,** but the **Slave** output can be changed only in the moment of the **falling edge** of the **Clock** signal. (Note: ignore **Enable** signal for now.)

Of course the best way would be to have the full schematics with **16 Accumulators**, but the available space on the panel allows only one. It must be switched with every new instruction, so that it displays the logic states of the register or memory location which is the current destination, and it may cause the unexpected switching of the output logic states of the Accumulator. For instance, when the button **Step** is first depressed in **Single Step** mode, the input logic states are transferred to the **Temporary Outputs** (outputs of **Master Flip-Flops**), but when the button is released, the same logic states would normally appear on **Accumulator Outputs**. However, the new instruction is read from the **Program Memory**, possibly with the new destination, and now the **Accumulator** represents the new register. That's why its logic states were unexpectedly changed.

This does not happen in **Direct** mode, at least in most cases, when the destination stays the same, so it is much easier to follow the data flow in the **Direct** mode.

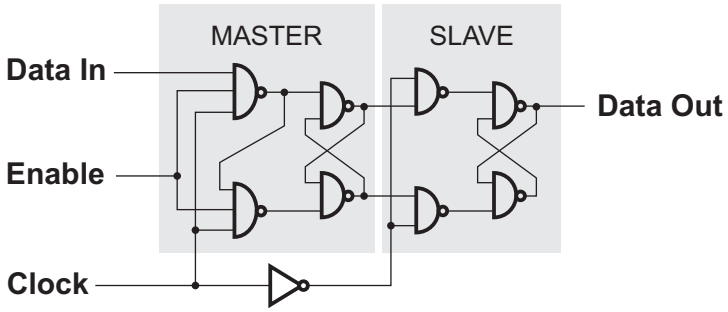
Indicators, buttons and connectors

STATUS register (1) and **FLAG logic** (2)

Status register, which contains **Overflow, Zero** and **Carry** flags, is probably the simplest, but among the most important parts of processor's hardware. Flags are kind of **decision-makers** in the program flow, and the **Carry Flag** is sometimes called the **1-bit Accumulator**.

Flags are propagated through the **Status Register** in a similar way as **Data Bits** are propagated through the **Accumulator**: there are three **Master-Slave Edge-Triggered D Flip-Flops**, which are triggered with the same **Clock** signal and at the same time as the **Accumulator Flip-Flops**. The **Status Flip-Flops** are not represented on the panel schematics in order to save space, but they are the same and triggered by the same **Clock** and **Inverse Clock** signals.

This is a good moment to say that there is the logic circuit which decides if the **Flip-Flop** will be clocked or not. Actually, the **Instruction Decoder** decides about that: some instructions have to keep the existing contents of the **Accumulator**, or individual **Flags**. Only when the **Accumulator** (to be more specific, the **addressed register**) is the destination of the operation, its contents should be clocked. If the current instruction does not affect **Flags** or **Accumulator** contents, the instruction decoder pulls the **Enable** signal low in the corresponding **Flip-Flop**(**s**), and thus locks the Flip-Flop state. The same is valid for every flag individually, as some instructions do not affect some flags, and they should be preserved safely in the **Flip-Flop**. The additional logic inputs, which disable the **Flip-Flop** clocking, is represented on the following schematics. There are the same **Latch Enable** inputs on the Accumulator Flip-Flops, as the contents of the Accumulator should be preserved in the case when it is not the destination (bit oriented instructions, program branching or compare instructions). Note that Enable inputs are not drawn on the simplified panel drawing.

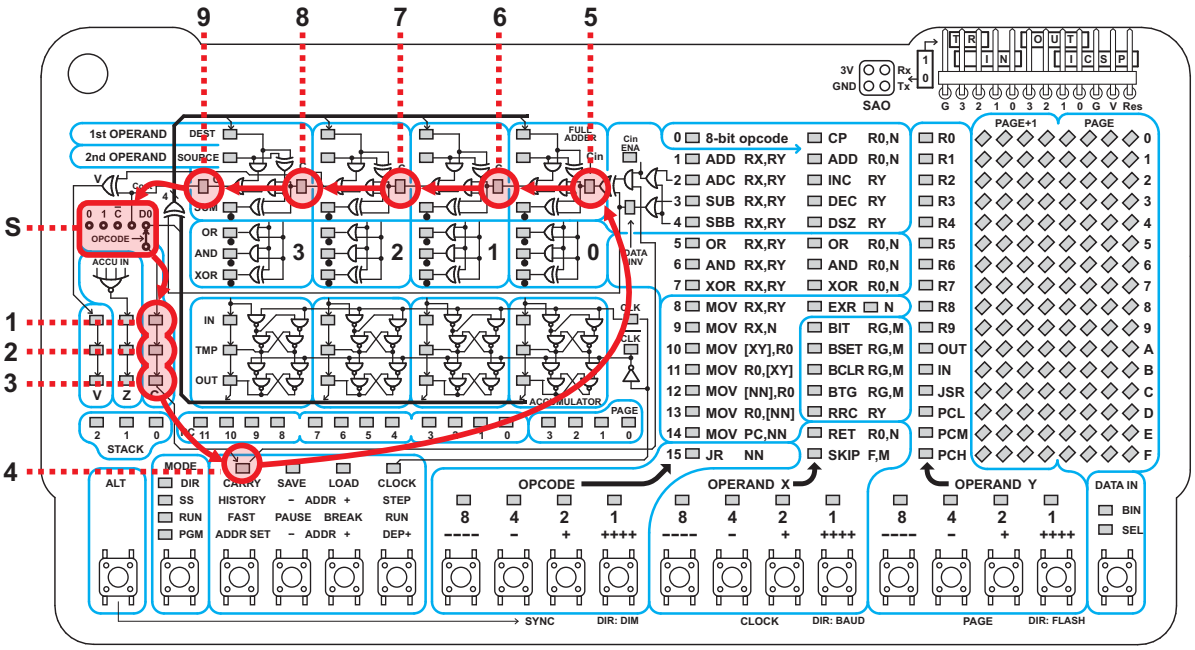


We have seen how simply the **Overflow Flag** is generated: a single **XOR** logic circuit detects the single-bit logic equality at the **Carry Input** and **Carry Output** signals of the last stage of the **Adder**. **Zero Flag** logic is also simple, as it only tests the **Accumulator** input for **Zero**. There is only one exception: instruction **BIT RG, M** sets or resets the **Zero Flag** depended on the tested bit state. Note that **Zero** condition always sets **Zero Flag** to **Non-Zero**, and **Non-Zero** condition resets it to **Zero**. This is valid not only for **BIT RG, M** instruction, but also in every other case. If the result is **Zero** (all bits are **0000**), the **Zero Flag** will be set (**1**), and if one or more bits in the result are set (**1**), the Zero Flag will be reset (**0**). In a few words, **Zero Flag = 1** means **Zero**, and **Zero Flag = 0** means **Non-Zero**.

Carry Flag can be a result of arithmetical or bitwise rotation. It can also be unconditionally **set** by the instruction **OR R0,N**, **reset** (**AND R0,N**) or **toggled** (**XOR R0,N**). The drawing on the following page, which represents **Carry** signal flow, is valid for **ADD/ADC/SUB/SBB** instructions (including **CP** also, which is the same as **SUB,** only without storing the result), but a similar flow could be drawn for **RRC** (**Rotate Right Through Carry**) also.

Carry Flag has a complex behavior, so it is represented in as much as nine indicators on the panel schematics. Please look the following page.

Indicators, buttons and connectors

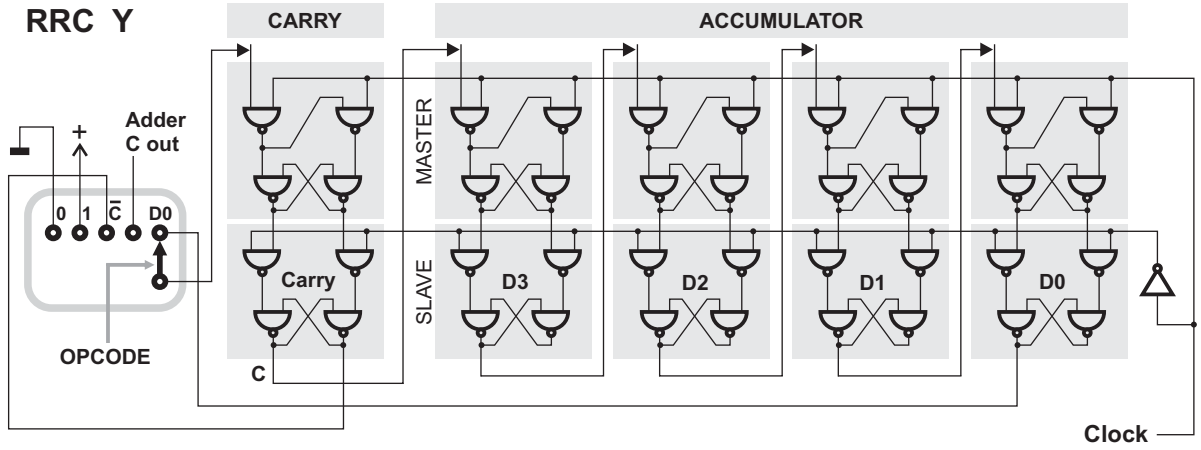


Nine indicators represent the same flag, and yet every one has the different meaning. After the selector **S**, which is under the control of the instruction decoder, the 1-bit content of the Carry Flag is fed to the input of **Carry Flip-Flop** (1) in the **Status** register. If the instruction is supposed to affect the **Carry Flag**, the **Clock** signal will first load the content to the **TMP** point (2) (**Master Flip-Flop** output), and then to the **Carry Output** (3) of the **Status** register.

At the same moment, the **Carry Flag LED** (4), which is in the **Command Buttons** field, fetches the same state. This indicator is not a part of the typical processor, but it is included in this model as it allows the user to modify the **Carry Flag** state manually, for experimenting. So you can watch interactively how **Carry Flag** affects the **Adder** states.

After passing through the simple logic, which inverts **Carry Flag** in the case of subtraction and turns it off if no Carry input is needed, there are **Intermedial Carry Flags** (6), (7), (8) between the **Adder** stages, before the final **Carry** (9) is generated.

In the case of the **RRC Y** instruction, the data flow of the **Carry Flag** is not represented here in detail, but there is the **D0** (**Data 0**) signal extracted from the internal **Data Bus**, which is fed to the rightmost contact of the **Selector S**. So the **D0** logic state is driven to the **Carry Flag**, and the rest of the data flow is represented on the following schematics diagram:



Stack Pointer (SP) (14)

This three-bit register is used to address the Data Memory where the **Program Counter** (**PC**) will be stored during the execution of the subroutine (writing to **JSR** General Purpose Register), and restored back to Program Counter at the execution of RETURN (**RET R0,N** instruction).

Program Counter contains **12** bits (**3** Data Memory locations), so one Stack position requires **3** nibbles for storing. When the subroutine is called (when the program writes a nibble in **JSR** General Purpose Register, on location **0x0C**), **PC** is stored at the Data Memory location **0x10+3×[SP]** (**low-order** address nibble first). Then the **SP** register is incremented by one, and **JSR**, **PCM** and **PCH** registers are copied to the **PC** (**JSR** is the **low-order** address nibble).

When the Return from subroutine (instruction **RET R0,N**) is executed, literal **N** is loaded to the register **R0**, then the **SP** is decremented by one, and contents of Data Memory from the three locations (the first one is 0x10+3×[SP]) is written back to the **PC**. Note that both **SP** and **PC** registers are not available directly to the user's program.

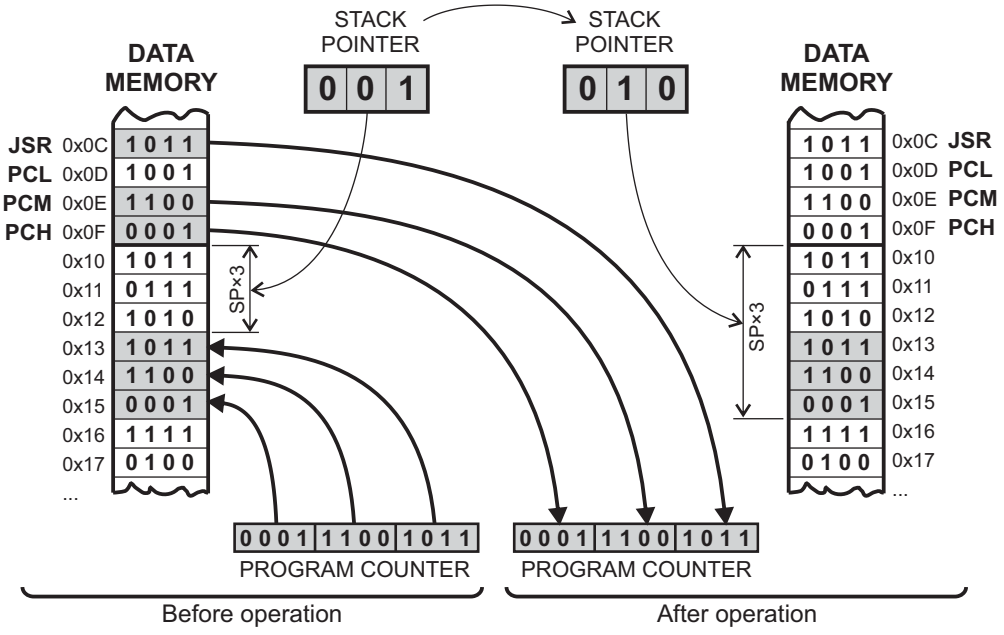
A total of **5** levels of Stack Pointer can be used in the program, which means that only the first **15** Data Memory locations from the **Page 1** will be used for the Stack storage. If the **SP** overflows to **110** (decimal **6**), which will happen if subroutines are called **6** times without executing **RET R0,N**, program execution will be halted and the **Error** condition will be indicated, so that the Stack indicator will blink at the value **110** (which is the attempted value when the error occurred). The Error condition should be cleared by pressing any key (the command assigned to the key will not be executed).

If more Returns (instructions **RET R0,N**) are executed than Calls (writing to the register **JSR**), register **SP** will be in the **underflow** condition. Program execution will be halted and the Error condition will be indicated, so that the Stack indicator will blink at the value **111** (decimal **-1** in signed notation). All register indicators and **Data Memory** matrix are still active, so the user can see the **PC Address** and other conditions under which the error occurred. The **Error** condition should be cleared by pressing any key (the command assigned to the key will not be executed).

When the unit is in the **RUN** mode, Stack Pointer is automatically cleared at every program **RUN** and program **Break**, and when the **Error** state is cleared. In **Single Step (SS)** mode, clearing **Data Ram** (performed when **Program Counter** is cleared by pressing **ALT-ADDR minus**), which also clears the **Stack**.

When entering **Single Step** (**SS**) mode, **Stack** restores the last value which it had in the **SS** mode.

Here is how **Stack Pointer** addresses **Data Memory** when the subroutine is called (when the value is written in the register **JSR**). The process is inverse when the instruction **RET R0, N** is executed.



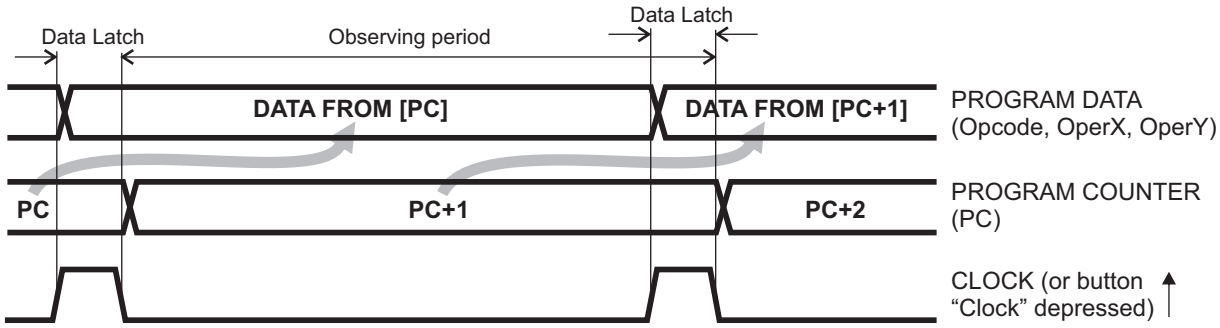
Program Memory Address (or Program Counter, PC) (18)

Program Counter is the **12-bit** pointer which keeps the address of the program word which is read from the **Program Memory** and executed by the processor.

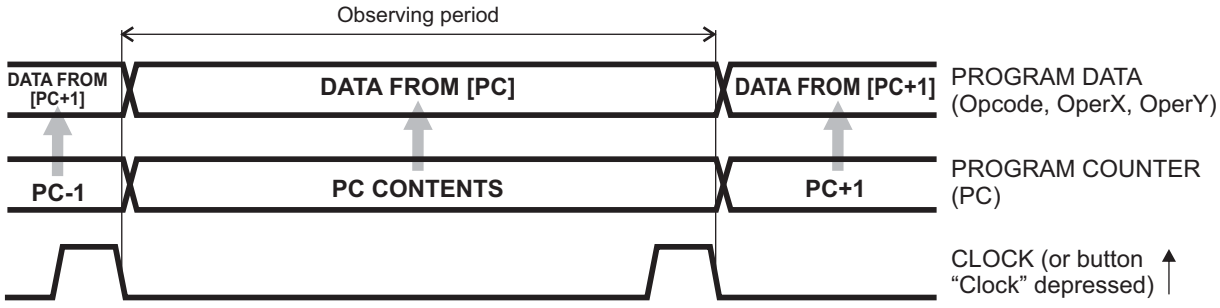
PC indicator (18) always displays the current state of the 12-bit **Program Counter**. The only exception is when the **ALT** button is depressed in the **History submode** of the **Single Step (SS)** mode. In that case (when in **SS** mode, **History** submode, and **ALT** depressed), **PC** indicator displays the 7-bit counter which denotes the depth of the History pointer. In other words, it shows how deep (how many backward steps) you are in the History "time machine" (how many Single Steps backwards you are watching registers and memory contents).

IMPORTANT NOTE: Immediately after reading the current program word from the Program Memory (which is performed automatically at the every clock pulse while the program is running), the **PC** should be automatically incremented by one, always pointing to the **NEXT** instruction, instead of the **CURRENT** one. That means that the **PC** is always ready to read the next instruction and it is a faster way to perform reading. That's why this method is used in normal processors.

Note: the following diagram is simplified. Please note the gray arrows.



That's what happens when we increment **PC** immediately after the instruction was read. If the program is running at a high speed, you wouldn't notice a problem, but in the slow modes (e.g. one instruction in two seconds, or even more in the Single Step (**SS**) mode, the **PC** indicator will always point to the **NEXT** instruction, instead to the **CURRENT** one. This means that you would have the address **[PC+1]** displayed on the **PC** indicator, and the data from the address **[PC]+0** displayed on the **Opcode, Operand X** and **Operand Y** indicators. This could be very confusing, and that's why another method was used in the badge, even if it is not consistent with all other processors:

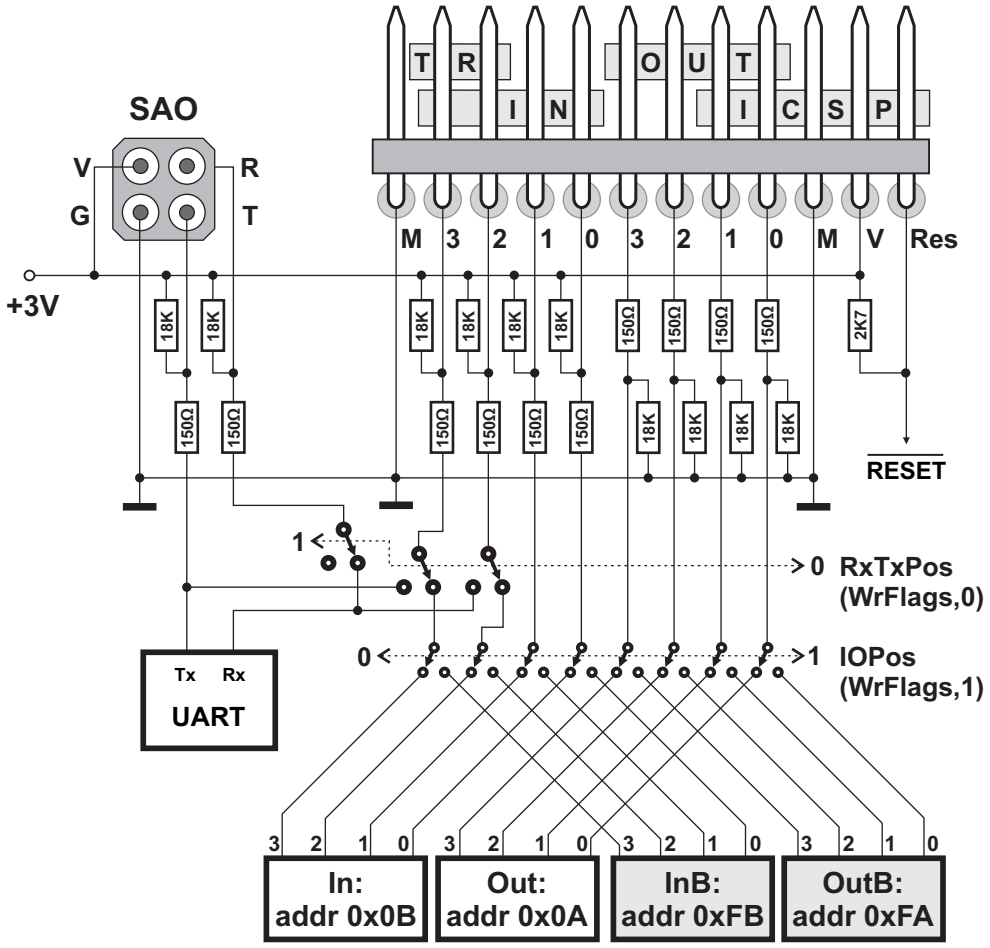


There is one more consequence of this non-consistency with the **PC** behavior of the real processor. Instruction **JR NN**, which modifies the current PC adding the 8-bit signed value **NN** to the **PC,** uses the **incremented PC value**, although the **current** (non-incremented) **value** is displayed on the **PC** indicator. For instance, if you want to perform the **dead loop** (infinite loop), you should not perform the instruction **JR 0**, but **JR -1**, as the program should jump relatively from the **next** location, and not from the **current** one. Reminder: **-1** is the **signed 2's complement** number, so it should be written as **1111 1111**. Since the Opcode for the instruction **JR** is **1111**, the instruction **JR -1** should be binary coded as **1111 1111 1111**.

Indicators, buttons and connectors

SAO (Shitty Add-On) connector (11)

SAO connector was recently standardized for simple badge add-ons. There are power supply contacts on this connector, but note that there is no **I2C** port, but **UART** terminals **Tx** and **Rx** instead.



Input / Output (I/O) connector (12)

Four **Input** and four **Output** ports are available in the **I/O** connector for hardware expansion. The same connector also offers five rightmost contacts for **ICSP** (**In Circuit Serial Programmer**) which can be used for programming and debugging of the firmware of the unit. Models from **PICKIT** and **ICD** series are pin-to-pin compatible with the connector.

There are two bits in the **SFR** register **WrFlag** which control the pin commutations of the **I/O** connector. When the first one, **RxTxPos (WrFlags,0)**, is set, **UART** is internally connected to the pins of the **I/O** connector (**Tx** is also available on the **SAO** connector, with the same signal and can be used in parallel with **Tx** signal on **I/O** connector). When the first one, **RxTxPos (WrFlags,0)**, is set, port inputs **3** and **2** read the **Tx** and **Rx** states and can not be used as general purpose **I/O** pins.

Bit **IOPos (WrFlags,1)** decides which registers will be used as **Output** and **Onput** registers. Default registers are **R10** and **R11** in the **General Purpose** group, on page **0x00** of the **Data Memory** (locations **0xFA** and **0xFB**). If this bit is set, all Inputs and Outputs are redirected to the **SFR** group, on page **0x0F** of the **Data Memory** (locations **0xFA** and **0xFB**), so registers **R10** and **R11** can be used as **General Purpose** registers.

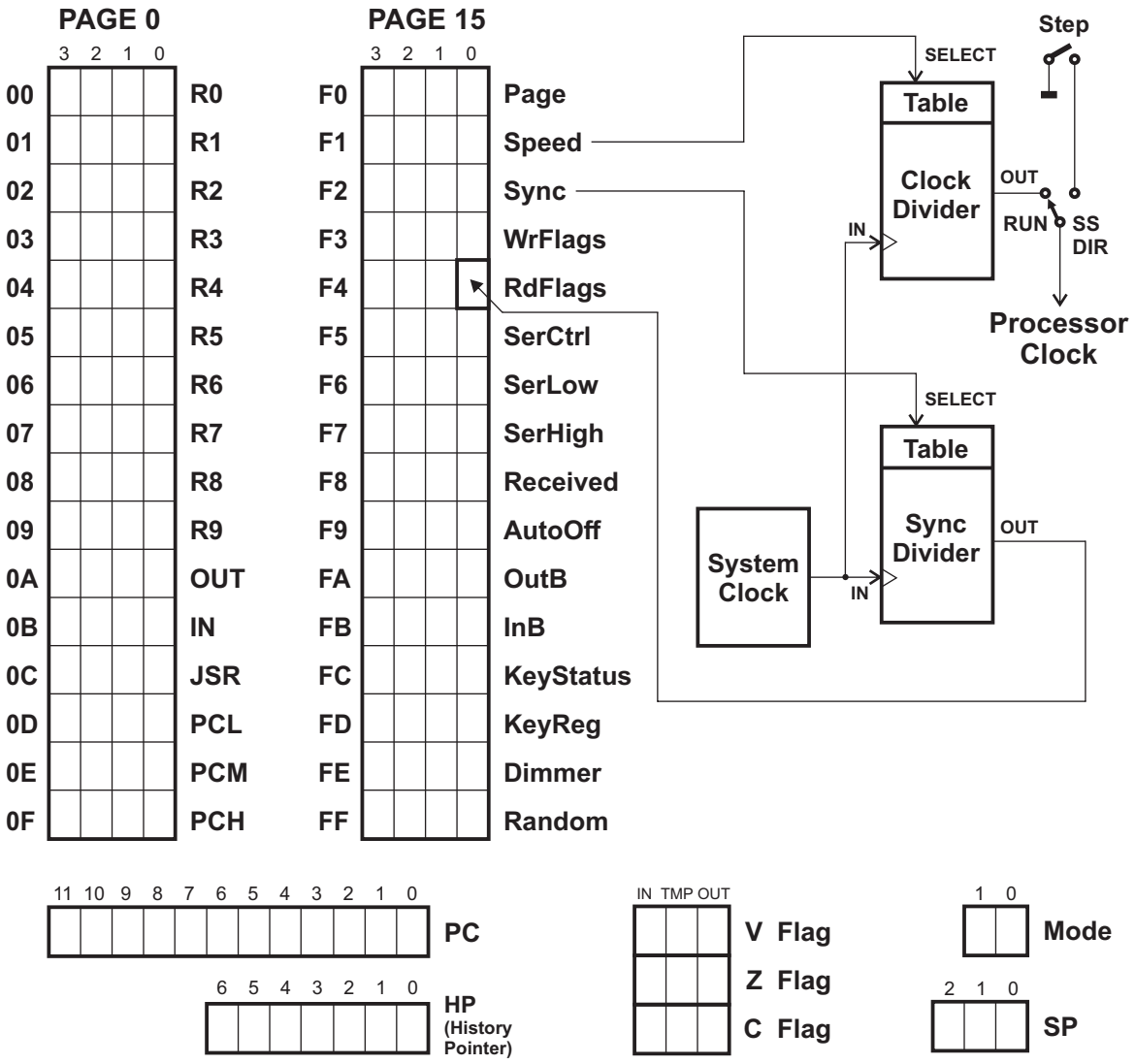
By default, **RxTxPos** and **IOPos** are reset. Default state is initialized at every Program **RUN** and **Break**. Note that Program **SAVE** and **LOAD** functions always use **Tx** and **Rx** pins on the **I/O** connector (not on **SAO** connector), regardless of the bit **RxTxPos** state.

Programmer's model

This is the hypothetical **4-bit** processor which is simulated by the firmware executed on the **16-bit** microcontroller **PIC24FJ256GA704**.

There is the **256-nibble** (**256×4**) **Data Memory**, which can not be expanded. Program is executed from the **4K** words (**4096×12**) **Program Memory**.

There are a total of **16 Main Registers** on **Page 0** (**0x00-0x0F**) of the **Data Memory**. Some of them (Registers **R0-R9**) are **General Purpose Registers**, and the rest of them (Registers **R10-R15**) are **Special Purpose Registers**, as they are dedicated to specific functions. This Register set can be reconfigured, so that two of the registers (**OUT** register, **R10** and **IN** register, **R11**) are moved to the **Special Function Register** group at **Page 15**, making free space for two more **General Purpose Registers**.



There are also a total of **16 Special Function Registers** (**SFR**), which are used to control processor's pseudo-hardware and perform special operations. All **Special Purpose Registers** are located on the **Page 0xF** (**0xF0-0xFF**) of the **Data Memory**. These registers are described in the manual **Special Function Registers**.

Stack Pointer (**SP**) is a **3-bit** register, not accessible directly to the user. **SP** memory is on the **Page 1 (0x10-0x1E)** of the **Data Memory**. It occupies **15** nibbles, which is enough for **5** levels of subroutines. The last nibble (**0x1F**) is not used, so it is available to the programmer. Also, if the programmer is sure that not all Stack levels will be used in the program, nibbles from that area can be used as a **General Purpose Data Memory**. Note that there are no **PUSH** or **POP** instructions.

Programmer's model

Program Counter (**PC**) keeps the **Program Memory Address**. As the **Program Memory** occupies **4096** words, **PC** register has **12** bits.

All internal registers and many internal logic states of the processor are displayed in real-time. Also, two selected pages (**16** nibbles each) are displayed on the LED matrix. However, all LED indicators are multiplexed, so some fast processor operations are subject to interference effect.

Internal clock frequency is user-selectable (and also selectable by the program) in **15** steps from **0.5 Hz** to **100 KHz**, and the **16th** clock frequency step is at the maximum speed, which is about **250 KHz**, but not guaranteed and not synchronized to the internal time base. Every instruction is executed in one clock cycle, so the maximum execution speed is about **0.25 MIPS**.

Another user-selectable (also selectable by the program) is the **Sync Timer.** It generates the internal heartbeat stream which has no impact neither on the hardware nor on the system firmware, but only sets the single bit in the **SFR** group (bit **0**, named **UserSync** in **RdFlags** register). User's program can test this bit as a flag at the handshaking manner, and thus synchronize the program flow to the uniform time periods. Available periods are arranged in **16 steps** from **1 ms** to **1 sec**.

There are three flags: **Overflow (V), Zero (Z)** and **Carry (C)**. Adder/Subtractor contains 4 Full Adder circuits, with Carry logic and Data inverters which enable full Add/Subtract operations for **unsigned** and **signed 4-bit** numbers with **Carry**.

Note: According to the schematic, data inverter for D0-D3 inverts the Destination bits only. However, in most cases it inverts Source bits, but there was no space for these bits here.

Every instruction has the length of **12 bits**. There is a table with all instructrions in this manual, and also the detailed description of every instruction with examples.

There are no **Long Jump** or **Call** instructions, so **Long Jumps** and **Subroutine Calls/Returns** are performed by writing a nibble in the Registers which are in the **Page 0**: writing to the register **PCL** performs the **Long Jump** to the location addressed by registers **PCL (0x13)**, **PCM (0x14)** and **PCH (0x15)**. Registers **PCM** and **PCH** must be pre-loaded with the desired high portion **(bits 11-4)** of the address.

Subroutine Call is performed in the similar manner, when the four lowest bits are written to the register **JSR (0x12)**. Registers **PCM** and **PCH** should be pre-loaded also. When the subroutine is called, the return address is written to the **Page 1** (**0x10-0X1E**) of the Data Memory, using the SP register, and then the SP register is incremented by 1. Address where the **12-bit Return Address** is stored, is calculated by the formula **0x10+(3×[SP])**.

When the **RETURN** instruction is executed, the process is reversed: Return address is read from the **SP** calculated address to the **PC,** and the **SP** is decremented. Also, **4-bit** literal number is loaded to the **R0** (which is on the address **0x00**). This can be used for lookup table read.

Not all instructions will initiate **Long Jumps** or **Calls** by writing to the **PCL** and **JSR** registers. Only the following instructions, which point to registers **PCL** or **JSR** as the destination, will do that:

MOV	**RX, RY**	(for calculated Jumps and Calls)
MOV	**RX,N**	(for simple Jumps and Calls)
INC	**RY**	(for repeated Table Reads)
DEC	**RY**	(for repeated Table Reads in reverse order)

MODES OF OPERATION

As it was mentioned before, there are four basic modes:

DIR	(Register addressing, directly executed instructions without memory usage)
SS	(Single stepping of the program from the Program Memory, temporarily editable)
RUN	(Program execution)
PGM	(Program writing or editing)

Mode **SS** also contains the submode **HISTORY**, which enables reviewing and analyzing of the last **127** program steps, with all processor signals and Data Memory contents.

There are also two special modes, used for hardware and firmware maintenance: **Test Mode** and **Bootload Mode**.

Programmer's model

ERROR PROCESSING

The only **Fatal Errors** which are possible at runtime, are the **Stack Errors**. Stack **Underflow** and Stack **Overflow** cause the unconditional program termination, with Stack indicator blinking, showing the illegally attempted state **110 (6)** on overflow, or **111 (-1)** on Underflow

INSTRUCTION SET

There is a total of **31** instructions, which are listed and described in detail in the manual **INSTRUCTION SET**. Only **11** instructions are available in **DIR** mode (manual **INSTRUCTION SET IN DIRECT MODE**).

Opcode and operand fields are divided in three **4-bit** groups, so that there are **4-bit** and **8-bit** opcodes (bits **11-4** or **7-4**) with one or two operands (only in a few special cases, used for **skip** and **bit manipulations**, there are **8-bit** opcodes and two **2-bit** operands). This strict and clear **Opcode/Operand** allocation inside the **12-bit Program Word space** greatly eases writing of programs in the pure machine language. Also, there are **4-to-16** decoders with LED indicators, which interactively display the program code disassembled in some way, making it much easier to read and write, without learning the instruction codes.

After the Data Memory Organization on the next page, there is a table with all instructions listed. After that, the detailed description of every instruction follows, with examples, coding schemes and flags affected.

DATA MEMORY

Data memory contains **256** nibbles (**256×4**), organized in **16** pages. Page **0 (0x00-0x0F)** contains a total of **16** main registers. The first **10** registers **(R0-R9)** are the General Purpose Registers, and the remaining **six (0x0A-0x0F)** are the **Special Function Registers**. Two functions (**Out** and **In**) which regularly occupy locations **0x0A** and **0x0B**, can be redirected to the **SFR** area on the last page, which allows free access to 12 General Purpose Registers. In that case, **Out** and **In** registers are at locations **0xFA** and **0xFB**.

The second page of the **Data Memory** is assigned to the **Stack.** It is the area where the **Return Address** will be written at every **Subroutine Call**. A total off **5** levels of **Stack** is allowed, and every **Return Address** takes **12** bits (**4** nibbles), so a total of **15** nibbles can be used for the **Stack**.

Page **14** can be used as the **Shaddow Register** area for a selected number of **Main Registers** from the page **0.** Instruction **EXR N** swaps the contents of pages **0** and **14**, in the length defined in the literal number **N**.

Page **15** is the Special Function Register (**SFR**) area, which contain **16** different registers with special functins. There is a detailed description of **SFR** in the manual "**Special Function Registers**".

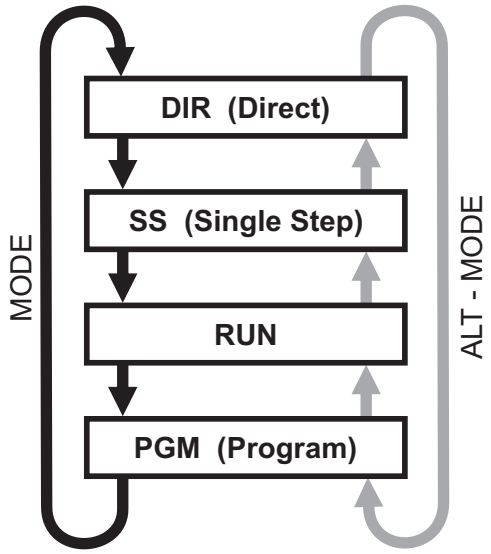
Every portion of **Data Memory**, in the length of two pages (**32** nibbles), can be interactively displayed on LED matrix. Register **PAGE** (which is in the **SFR** area at the address **0xF0**) determines which page will be displayed. The selected page is on the right halve of the display, and the next page is on the left. If the selected page is **15**, then the next page, displayed on the left halve, will be **0**.

Direct (DIR) Mode

MODE button and indicators (16)

There are four main modes: **DIR** (Direct), **SS** (Single Step), **RUN** and **PGM** (Program) mode. They are selected sequentially by pressing the **Mode** button. At every press, the next mode is set in increasing order. When **ALT** button is depressed, **Mode** button selects the mode in reversed order.

Every button from the **Command Group** (17) has the different function, which is depended on the current **Mode**. The description of available modes is on the following pages.

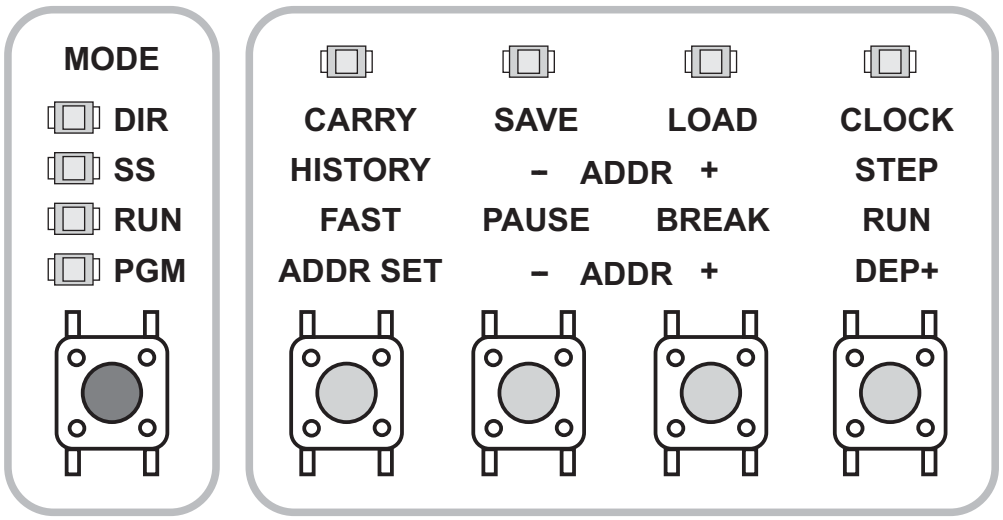


Direct writing and execution of instructions using **Register** addressing. Also program **Save** and **Load**.

Program execution from the **Program Memory** with manual modification and single stepping.

Program execution from the **Program Memory** driven by the internal adjustable **Clock**.

Program editor for direct writing to the internal **Program Memory**



Direct (DIR) Mode

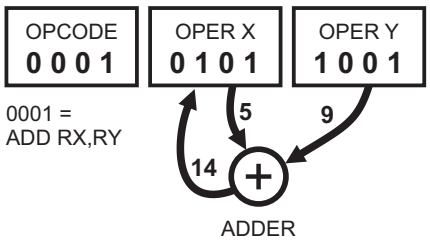
DIR Mode can be used for experimenting with different parts of the processor core: **Accumulator**, **Register X**, **Register Y**, **Adder/Subtractor**, **Logic Group**, **Flags** and so on. Most instructions can be executed by pressing the button **Clock**, but there is no indirect addressing mode, as registers **Operand X** and **Operand Y** contain only literal values. Data Memory is not accessible, which also means that General Function registers and Special Function registers do not exist in **DIR** mode. **I/O** connector is also not accessible. Programs written in **Program Memory** have no effect in **DIR** mode. The same is valid for **Program Counter (PC)**, **Stack Pointer** or **Page** registers.

All flags are normally active in the **DIR** mode.

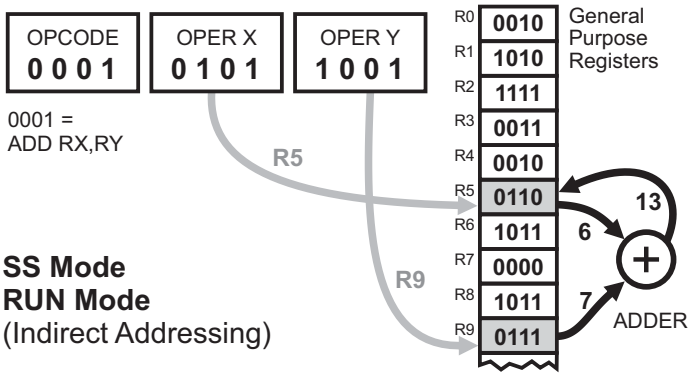
Instructions which communicate with the Data Memory cannot be executed in **DIR** mode. If the non-existing instruction is selected, **LED** which points to the instruction in the vertical decoded fields, will blink and the **Clock** button will not be active.

Registers **RX** (available in the **Operand X** field) and **RY** (in **Operand Y** field) are used as the two directly accessible 4-bit registers, and they do not point to the first page of the Data Memory, like in all other modes. If register **RX** or **RY** is the destination, the result will be written directly to the register **RX** or **RY**.

Here's an example: if the instruction **ADD RX,RY** (**0001**) is selected in the **Opcode** field, in the **DIR** mode it will be executed so that value displayed in the **Operand Y** field is added to the value displayed in the **Operand X** field and, when the button **Clock** is pressed once, the result (which is indicated on the **Adder** output and the **Accumulator** input) will be written to the register **RX** (indicators in the **Operand X** field).



Direct Mode
(Register Addressing)



SS Mode
RUN Mode
(Indirect Addressing)

In **SS** and **RUN** modes, processor will operate with contents of the General Purpose registers in **Data Memory**, which are addressed with **RX** and **RY** registers. For instance, if the value of **Operand X** is **0101** (decimal **5**) and the value of **Operand Y** is **1001** (decimal 9), processor will read values of registers **R5** and **R9**, which are in the **Data Memory** at the addresses **0x05** and **0x09**, adder will calculate the sum and write the result to the register **R5** (**Data Memory** address **0x05**).

User Program can be saved or loaded in **DIR** mode. Media available for program storing is the internal **Flash** memory, which can store up to **15** user's programs, or the external store unit which communicates through **UART** (**Universal Asynchronous Receiver-Transmitter**). Transmit and Receive terminals are with **3V** logic levels, so the best way to implement the serial communication with the laptop or desktop computer is the **USB/Serial** converter. Don't forget to cross Rx and Tx conductors, but also to **leave the + terminal from the converter unconnected**! Two power sources should never be connected in parallel, as the consequences could be unpredictable.

The same connection with the computer enables program sharing and loading of externally generated programs (assembler or some other way to generate the program code).

Although the **Display Matrix** is not active in **DIR** mode, there are two exceptions. The first one is displaying of **Program Version/Revision** and **Release Da**te, which is performed after the **Master Reset** (by removing and reinserting batteries or shorting pins **Res** and **G** on the **I/O** connector). This will be displayed only once after the Master Reset, and it will be cleared when any button is pressed.

Direct (DIR) Mode

Another exception of the **Display Matrix** function in **DIR** mode is displaying the **Flash** occupancy before the program saving or loading. To see **Flash** occupancy in **DIR** mode, just keep the **ALT** button depressed. Every pixel in this case represents the occupancy of **512** program words.

	Carry	Save	Load	Clock
	Toggle Carry Flag	Send Program Memory to Serial Port	Load Program Memory from Serial Port	Master Clock source
ALT		Save Program Memory to selected Flash	Load Program Memory from selected Flash	

	Opcode	Operand X	Operand Y	Data In
	Instruction Opcode (bits 11-8)	Direct Operand X or Opcode (bits 7-4)	Direct Operand Y (bits 3-0)	Toggle between BINary and SELect mode
ALT	Dimmer level select	Baud Rate select	Flash portion select for Save / Load	

DIR Mode: Button CARRY

Indicator **Carry** in the **Command Group** (17) is automatically updated at every instruction execution, but user can toggle it by pressing the button **Carry** under the indicator. It can be helpful in experimenting and watching how **Carry Flag** affects execution in certain processor core sections, especially in **Adder/Subtractor**.

DIR Mode: Button SAVE

Saving to the external unit through the Serial Port is performed by simply pressing **SAVE** button in **DIR** mode.

Note that the unit has no feedback from the external unit, and it will send the program immediately to the **Serial Port**, even if there is no connection. So it is essential that everything is prepared before the button **SAVE** is pressed, and that the external unit already expects data from the port. This is valid also if the program is shared directly between the two equal units.

Here are the rules for serial **Save/Load**:

- Serial Port settings: **9600, N, 8, 1** (not affected by **SFR** setting in **SerCtrl** register)
- Hardware considerations: **3V CMOS** Logic levels (**Tx/Rx** on **I/O connector** only)
- **Tx/Rx** on **SAO connector** can not be used for program saving or loading
- Software protocol (in Hexadecimal form):

1. Header **6** bytes:	**00 FF 00 FF A5 C3**
2. Program length **2 bytes** (in 16-bit words, Low byte first):	**NN NN**
3. Program **NN 0N×Program Length** (Low first):	**NN 0N, NN 0N, NN 0N...**
4. 16-bit Checksum **2 bytes** (items 2 and 3 only, Low first):	**NN NN**

Since the program data is contained of **12-bit** words, writing format is **16 bits** for every word, but the upper nibble (bits **15-12**) is always dummy **0**. Header is a simple 6-byte string, and all other items are 16-bit numbers in **Little Endian** order (Least Significant Byte first).

However, if there is nothing to save (if the whole Program Memory is empty), indicator **SAVE** will blink and no data will be sent to the Serial Port. In that case, any button will quit blinking and return the unit to the previous state.

During program saving, the taskbar appears on the three vertical decoded indicator bars.

If the program has to be saved to the internal **Flash** memory, **ALT-SAVE** has to be typed (hold **ALT** while pressing **SAVE**). But before that, a few steps should be performed. First, select the address of the portion of Flash memory where the program will be saved. There are a total of **15** available portions (the 16th one has a special function), and the selection of the desired one should be performed by pressing and holding the **ALT** button, while selecting the portion by pressing buttons in the **Operand Y** field. The helpful feature is that the Flash occupancy is displayed on the Display Matrix, so you just have to match the vertical indicator bar next to the desired portion. When you are finished with the Flash portion choice, just press **SAVE** while you are still holding **ALT**.

Writing to the Flash memory is performed much faster than writing to the **Serial Port**, so no taskbar is displayed here. As the master processor, which drives the unit, halts the program execution during programming, display matrix refresh is inhibited and all LEDs (except **SAVE**) are switched off for a fraction of a second. This is quite normal.

There is no **Undo** option, and no additional verification before the program saving, and the previous contents of the addressed **Flash** portion will be overwritten unconditionally, so please take good care when selecting the target portion of the **Flash** memory.

DIR Mode: Button LOAD

Loading is performed similarly to saving, but in the inverse manner. Command for loading from the external computer or some other unit via **UART** is simply pressing **LOAD**, and when the data starts flowing and the internal program memory loading data, the same taskbar appears on the three vertical bars, but this time in the inverse direction, upside down, suggesting the data flow from the **I/O** connector to the unit. If the **LOAD** button is pressed unintentionally in **DIR** mode, the process can be stopped by pressing any key, and if the valid header is not received through the serial port, the contents of the internal program memory will not be destroyed.

Command for program loading from the internal Flash memory is **ALT-LOAD**, but before that the same selection of the **Flash** portion should be performed. Every time prior to the program loading from the internal Flash, automatic safety writing of the current **Program Memory** contents to the location **15** is performed. So if you loaded the program from the **Flash** memory unintentionally (for instance, if you intended to perform **Save**), the contents of the internal **Program Memory** is wiped out, but there is a spare copy in the location **15** and we can load the program from it. The only thing that can be really destructive, is if you perform unwanted loading twice, as the new safety writing, which is performed automatically at the **LOAD** command, could wipe out the **Flash** location **15**, this time without the spare copy.

It means that Flash location **15** has the special function and can not be used for program storage. Command to write to this location (**ALT-SAVE** with location **15** selected) will be ignored by the system, but loading from location **15** will be executed normally (this time without automatic spare copying).

Here are the rules for serial **Save/Load**:

- Serial Port settings: **9600, N, 8, 1** (not affected by **SFR** setting in **SerCtrl** register)
- Hardware considerations: **3V CMOS** Logic levels (**Tx/Rx** on **I/O connector** only)
- **Tx/Rx** on **SAO connector** cannot be used for program saving or loading
- Software protocol is same as for LOAD command

DIR Mode: Button CLOCK

When the **Clock** button is pressed in DIR mode, the result of the current operation and **Flags** are latched in the **Master** Flip-Flops of the **Accumulator** and the **Status** register, respectively (note that every instruction enables or disables latching for the Accumulator and every Flag separately, depending on the instruction context).

Direct (DIR) Mode

Note: There are actually two Accumulators in DIR mode, one of them is **Register X** (available and displayed in the **Operand X** field) and the another one is **Register Y** (available and displayed in the **Operand Y** field). The one that is displayed is actually the destination of the operation. If there are two operands in the instruction, it will be **RX**, and if there is only one, it's **RY**. Switching of the two of them (which occurs when the Clock button is released and the next instruction does not use the same destination) can sometimes cause confusion.

After the **Clock** button is released, **Slave** Flip-flops in the **Accumulator** and the **Status** register perform the final latching and the result (if any) is safely stored in **Slave** Flip-flops. The new value of the destination register is written to **Rx** or **RY**, the new instruction is executed and the stage is ready for the next pressing of the **Clock** button.

Single Step (SS) Mode

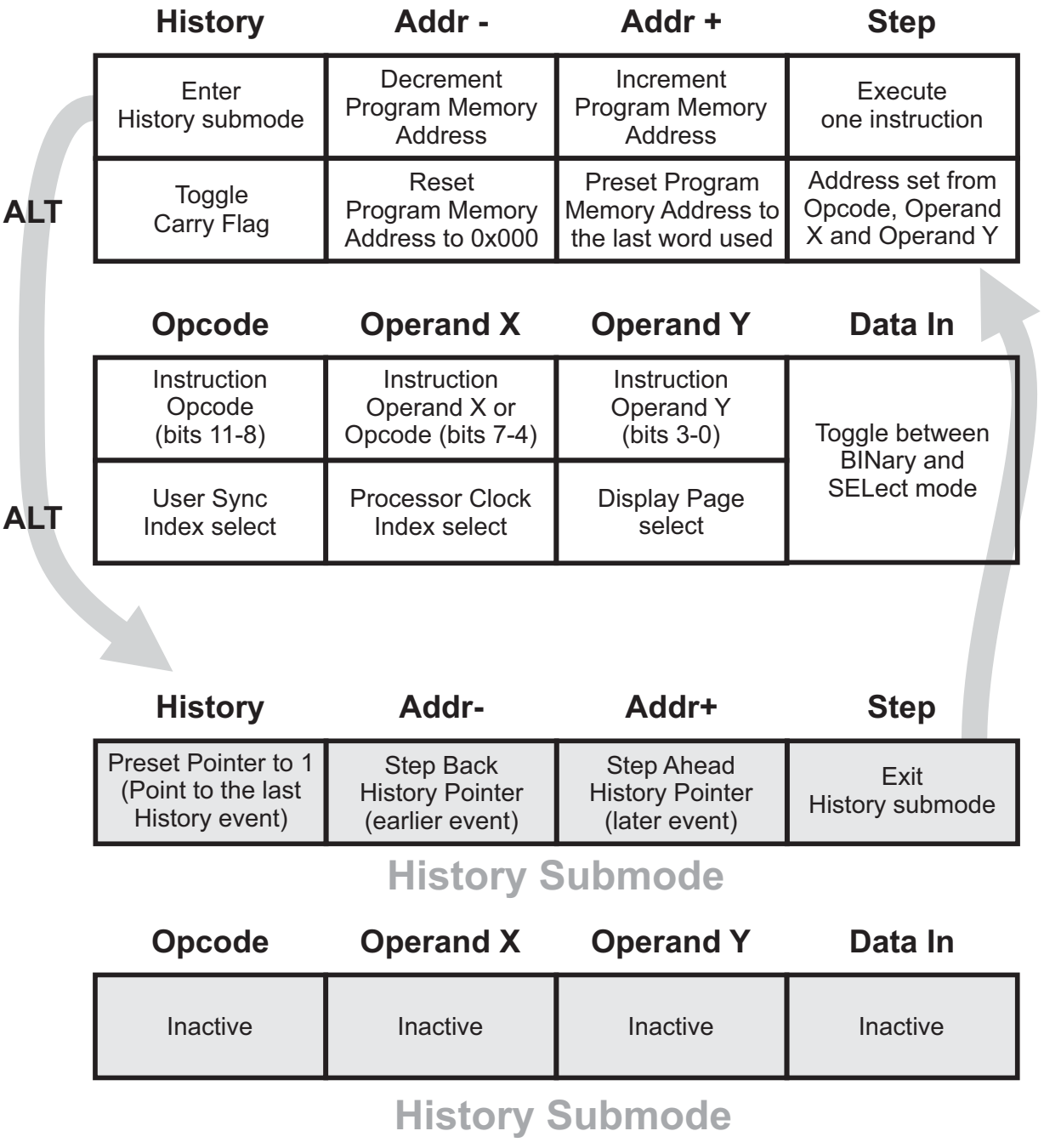
Single Step (SS) Mode

Single Step (SS) Mode supports manual stepping through the program written in the **Program Memory**. Modification of the contents of every program word is possible, but it is valid only once, for direct execution only (which is initiated by pressing "**STEP**"). Permanent modification of **Program Memory** is possible in **PGM** mode only.

Single Step (SS) Mode also contains the extra **HISTORY** submode, which allows reviewing of the last **127** steps performed in **Single Step (SS)** mode.

Display Matrix in **SS** mode shows the contents of **Data Memory** on the selected **Page**, but while the button **ALT** is pressed, **Page 0** (right) and **Page 1** (left) are unconditionally displayed.

Indicator **Carry** shows the **Carry Flag** state, but it cannot be modified manually in **SS** mode.



Single Step (SS) Mode

Single Step (SS) Mode: Button HISTORY

Button **HISTORY** is used to enter **History** submode, which allows you to review the last **127** steps executed in **SS** mode. All visible indicators show the state of registers and logic levels when the event was executed and automatically recorded.

Data Memory contents of the page which was selected is also displayed, and when the button **ALT** is pressed in **History** submode, contents of Pages **0** and **1** (which were valid when the event was recorded) is also displayed.

Program Counter (PC) indicator shows the **Program Memory Address** of the recorded event, and when **ALT** is pressed, it displays the **History Pointer** state. That means that you can see how "deep" you are in **History** steps (how old is the displayed event, in executed steps).

Indicator **SS** blinks, signifying that the unit is in the **History** submode. Pressing **ADDR-** or **ADDR+** decrements or increments the **History Pointer**. When pressing of **ADDR-** or **ADDR+** causes the corresponding indicators to turn on, that means that pointer reached the start **(1)** or the end **(127)** of the History buffer.

The general rule in History mode is that you can see all program parameters and registers, but you can't modify anything.

Pressing the button **STEP** exits **HISTORY** submode and returns back to the normal **SS** mode.

If **ALT** is also depressed, then button **HISTORY** only toggles the **CARRY** flag.

Single Step (SS) Mode: Button ADDR -

Button **ADDR-** in **SS** mode decrements the 12-bit **Program Memory Address** pointer by 1.

If **ALT** is also depressed, then button **ADDR-** resets Program Memory Address to **0000 0000 0000**. Also, the whole **Data Memory**, **Stack Pointer** and **Page** register are cleared to zero.

If the current Program Memory Address is **0000 0000 0000**, decrementing will wrap the new value to **1111 1111 1111**, which is the last program word in the memory.

In History submode, button **ADDR-** steps back the History pointer (earlier event).

Single Step (SS) Mode: Button ADDR +

Button **ADDR+** in **SS** mode increments the 12-bit **Program Memory Address** pointer by 1. If **ALT** is depressed, then button **ADDR+** sets Program Memory Address to the last program word used (the last one which does not contain value **0000 0000 0000**).

If the current Program Memory Address is **1111 1111 1111**, which is the last program word in the memory, incrementing will wrap the new value to **0000 0000 0000**.

In History submode, button **ADDR+** steps ahead the History pointer (later event).

Single Step (SS) Mode

Single Step (SS) Mode: Button STEP

When the button **STEP** is pressed in SS mode, one program step (execution of the current instruction) is performed. When the button is depressed, the result of the current operation and **Flags** are latched in the **Master** Flip-Flops of the **Accumulator** and the **Status** register, respectively (note that every instruction enables or disables latching for the Accumulator and every Flag separately, depending on the instruction context).

After the **STEP** button is released, **Slave** Flip-flops in the **Accumulator** and the **Status** register perform the final latching and the result (if any) is safely stored in **Slave** Flip-flops. The new value of the destination register is, in most cases, written to the destination address (which depends on the instruction), then the **Program Counter** is incremented by **1** (or preset to the new value if the instruction caused program branching). Then the contents of the **Program Memory**, addressed by **Program Counter**, is read, latched and displayed by the indicators in **Opcode, Operand X** and **Operand Y**. The execution of the new instruction is simulated and all indicators show the correct state of the internal registers and data paths, but storing the result of the instruction has to wait for the new **STEP** cycle.

At this moment, when the system is waiting for the new **STEP** command, you can modify every bit in the instruction, and watch interactively how the processor registers react to the new values. The data which is supposed to be written at the destination, appears at the 4-bit **Accumulator** inputs, and the flags at the **Status Register** inputs (upper row). However, they will not be written to the destination if the instruction does not allow that. Some instructions don't write data to the destination (e.g. **CP Y** or **SKIP F, M**) and many instructions affect only some flags, if any. That's why some flags are shifted down in the register when the **STEP** button is pressed, and some are not. You can see which are affected by which instructions on the table at the back (bottom layer) of the unit PCB, or in the instruction list (PDF file "**INSTRUCTION SET**").

Every modification of the instruction is taken into account, and when the **STEP** button is pressed, the instruction in **Opcode, Operand X** and **Operand Y** registers will be executed, even if the **Program Memory** contains some other data. But the modification will be valid for only one execution, as the **Program Memory** contents were not modified. The only mode which has the right to affect the Program Memory contents is the **PGM** mode.

Prior to every instruction execution in **SS** mode, which happens when the button **STEP** is released, all machine states, flags, registers and two portions of the **Data Memory** (the currently displayed, and the portion of **Page 0** and **Page 1**) are loaded into the **127**-block **History Buffer**. The contents of this buffer will be used if the **History** submode is invoked.

When **ALT** is depressed, button **STEP** does not execute code in **Opcode, Operand X** and **Operand Y**, but loads the contents of these registers to the **Program Counter** (same as **Addr Set** command in **PGM** mode). It is a fast way to preset **PC** to some predetermined value.

RUN Mode

RUN Mode

RUN Mode is used to control the execution of the program written in **Program Memory**. When the **RUN** button is pressed in **RUN** mode, most registers and pointers (**Stack, Page, Clock, Sync, WrFlags**) are cleared, and some of them are preset to default values (**Dimmer** to maximum brightness, and **SerCtrl** to **0011** binary, so the **Baud Rate** is set to **9600**). The whole **Data Memory** is cleared, and then the execution starts from the **Program Memory Address 0000 0000 0000**. So the program is always executed with the known initial values, and the further state of all control bits and variables is at the user's program responsibility.

After the program is terminated (which will happen if the processor detected the **Stack Error**, or when button **BREAK** is pressed), the same register clearing process is performed again.

Buttons in **Opcode / Operand X / Operand Y** fields are inactive in **RUN** mode, so the code can not be affected. However, the same buttons are active with **ALT** pressed, so it is possible to modify **Sync, Clock** and **Page** parameters. If the program is running, it has to be paused first to perform the modification of **Sync, Clock** and **Page**.

Note that **Sync, Clock** and **Page** are regular **SFR (Special Function Registers)** and that they can be preset at any time under the program control.

	Fast	Pause	Break	Run
	On/Off Toggle 10× Faster Clock and Sync (if possible)	Program Execution Pause / Resume	Terminate Program Execution	RUN Program From Program Memory

Active only at runtime

	Opcode	Operand X	Operand Y	Data In
	Inactive	Inactive	Inactive	Toggle between BINary and SELect mode
ALT	User Sync Index select	Processor Clock Index select	Display Page select	

RUN Mode: Button FAST

Button **FAST** is used for switching between normal and fast (**10×**) execution mode. When the **FAST** mode is activated, indicator **RUN** in **MODE** field blinks.

In **FAST** mode, indexes in **SYNC** and **CLOCK** variables are temporarily modified to point to the higher speed. If normal settings are already at the maximum speed or close to the maximum, **FAST** mode will have no or little effect.

Pressing the button **FAST** has no effect if the program is not running.

RUN Mode

RUN Mode: Button PAUSE

Button **PAUSE** stops execution temporarily. While the **PAUSE** mode is active, indicator **PAUSE** blinks. In **Pause** mode, it is possible to modify variables **Sync, Clock** and **Page** by holding button **ALT** and pressing buttons in the **Opcode, Operand X** and **Operand Y**. There is a text under these buttons to helps orientation and serves as the command reminder.

Exit from **PAUSE** is performed by pressing the button **RUN**. Program execution will be performed normally. Pressing the button **BREAK** in Pause mode quits the program execution.

Pressing the button **PAUSE** has no effect if the program is not running.

RUN Mode: Button BREAK

The only way to quit the program execution regularly is to press the button **BREAK**. It is also possible to stop the PAUSE mode and to quit the program execution with the button BREAK.

When the **BREAK** button is pressed in **RUN** mode, most registers and pointers (**Stack, Page, Clock, Sync, WrFlags**) are cleared, and some of them are preset to default values (**Dimmer** to maximum brightness, and **SerCtrl** to **0011** binary, so the **Baud Rate** is set to **9600**). The whole **Data Memory** is cleared, and the unit is ready to restart the same or load the new program, or to change the mode.

Pressing the button **BREAK** in RUN mode has no effect if the program is not running.

RUN Mode: Button RUN

Button **RUN** in **RUN** mode starts the program execution from the **Program Memory**, starting from the address **0000 0000 0000**.

When the **RUN** button is pressed, most registers and pointers (**Stack, Page, Clock, Sync, WrFlags**) are cleared, and some of them are preset to default values (**Dimmer** to maximum brightness, and **SerCtrl** to **0011** binary, so the default **Baud Rate** is set to **9600**). The whole **Data Memory** is cleared. So the program is always executed with the known initial values, and it can adjust them dynamically, as needed.

Program (PGM) Mode

Program (PGM) Mode

Except for the **LOAD** command, **Program (PGM) Mode** is the only mode which allows modification and writing to the **Program Memory**. The regular procedure is to set the **12-bit Program Word** (preset and displayed in the **Opcode, Operand X** and **Operand Y** fields), and then, assuming that the **Program Counter (PC)** points to the desired address, press button **DEP+** (**Deposit** with post-increment). At that moment, the **12-bit Program Word** will be written to the internal **Program Memory** and the **Program Counter (PC)** will be incremented by **1**.

	Opcode	Operand X	Operand Y	Data In
	Instruction Opcode (bits 11-8)	Instruction Operand X or Opcode (bits 7-4)	Instruction Operand Y (bits 3-0)	Toggle between BINary and SELect mode
ALT→	User Sync Index select	Processor Clock Index select	Display Page select	

	Addr Set	Addr-	Addr+	Dep+
	Address set from Opcode, Operand X and Operand Y	Decrement Program Memory Address	Increment Program Memory Address	Write the current word, overwriting the old one, increment PC
ALT→	Delete the current word (move all subsequent words down)	Reset the Page and Program Memory Address to 0x000	Preset Program Memory Address to the last word used	Write the current word, moving (pushing) all subsequent words up
ALT / Both buttons→	Clear all Memory,**Data Memory** and most registers (Note: **NO UNDO**!)			

Program (PGM) Mode: ADDR SET

When the button **ADDR SET** is pressed in **PGM** mode, the contents of **Opcode, Operand X** and **Operand Y** registers are copied to the **Program Counter (PC)**. This is a convenient way to preset the desired **Program Address**. Note that **Opcode, Operand X** and **Operand Y** fields are not used for **Program Data**, like in all other cases, but for **Program Address**.

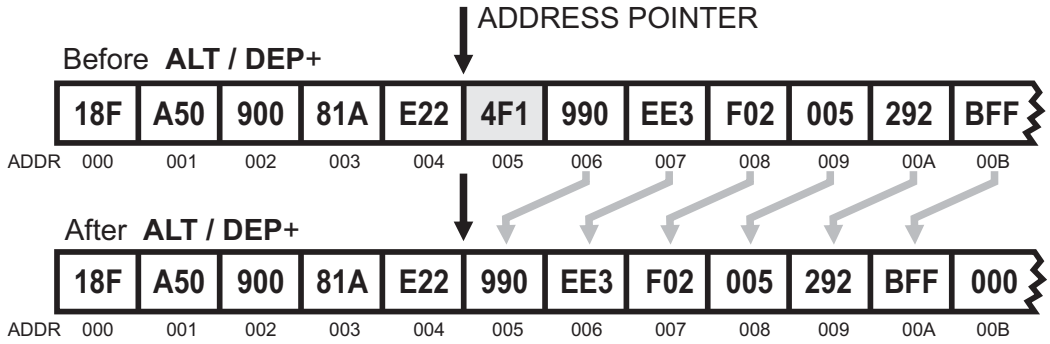
Note that every time when the **Program Address** is modified, the contents of the addressed memory location is transferred to the **Opcode, Operand X** and **Operand Y** fields.

Program (PGM) Mode

Program (PGM) Mode: ALT / ADDR SET

If **ALT** is depressed, command **ADDR SET** deletes the current Program Word and moves all the subsequent words in the Program Memory one place down, thus overwriting the current memory location. The **Program Address Pointer** is unchanged. There is no **UNDO**.

Here is an example:



Note: Command **ALT / ADDR SET** moves all subsequent program words one position down, which typically means that all symbol names should be modified. This has no effect only if the current instruction pointer is near the end of the written program and there are no symbol names behind it.

Program (PGM) Mode: ADDR-

Button **ADDR-** in **PGM** mode decrements the 12-bit **Program Memory Address** pointer by 1.

If the current Program Memory Address is **0000 0000 0000**, decrementing will wrap the new value to **1111 1111 1111**.

Note that every time when the **Program Address** is modified, the contents of the addressed memory location is automatically transferred to the **Opcode, Operand X** and **Operand Y** fields.

Program (PGM) Mode: ALT / ADDR-

If **ALT** is depressed, then button **ADDR-** resets Program Memory Address to **0000 0000 0000**. Also, the whole **Data Memory**, **Stack Pointer** and **Page** register are cleared to zero.

Note that every time when the **Program Address** is modified, the contents of the addressed memory location is automatically transferred to the **Opcode, Operand X** and **Operand Y** fields

Program (PGM) Mode: ADDR+

Button **ADDR+** in **PGM** mode increments the 12-bit **Program Memory Address** pointer by 1..

If the current Program Memory Address is **1111 1111 1111**, which is the last program word in memory, incrementing will wrap the new value to **0000 0000 0000**.

Note that every time when the **Program Address** is modified, the contents of the addressed memory location is automatically transferred to the **Opcode, Operand X** and **Operand Y** fields.

Program (PGM) Mode: ALT / ADDR+

If **ALT** is depressed, command **ADDR+** sets the Program Address Pointer (current program word) to the last word used in program (the last word which is not 0x000).

Program (PGM) Mode

Program (PGM) Mode: ALT / ADDR- / ADDR+

If **ALT** is depressed, pressing **ADDR-** and **ADDR-** simulteneously clears the whole **Program Memory, Data Memory** and most registers. There is no **UNDO**.

Program (PGM) Mode: DEP+

Button **DEP+** in **PGM** mode is used to store the contents of the **Opcode, Operand X** and **Operand Y** fields to the program memory, at the location defined by the **Program Memory Address** , overwriting the previous contents (no **UNDO**). After storing the **Program Word** to the **Program Memory,** the **Program Counter (PC)** is automatically incremented by **1**. Then the contents of the new location (addressed by the new value of **PC**) is automatically transferred to the **Opcode, Operand X** and **Operand Y** indicators.

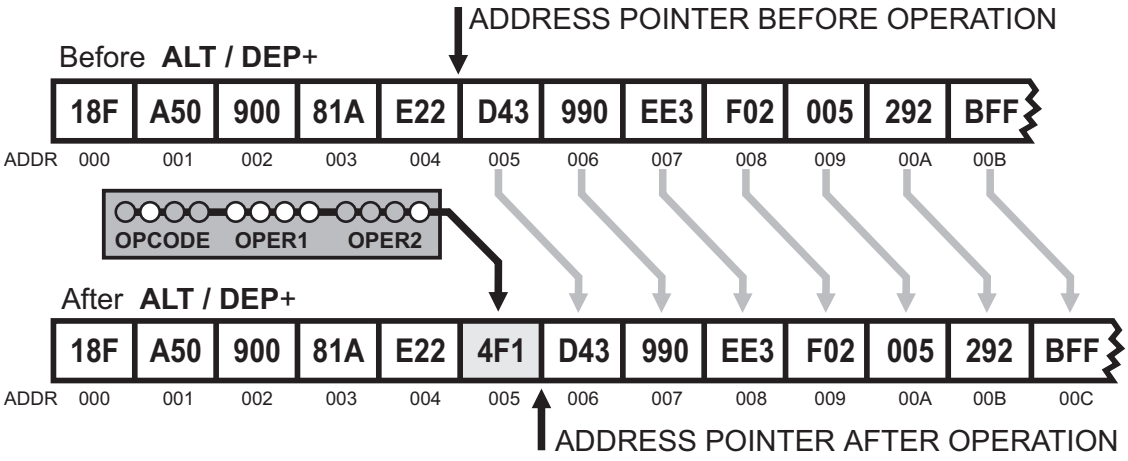
This means two things, which should be kept in mind during program writing. First, if the **DEP+** button is repeatedly pressed in **PGM** mode without modifying the **Program Word** contents, it will not affect the contents of the **Program Memory**, as every location is first read and stored to the **Opcode, Operand X** and **Operand Y** registers, and then rewritten to the memory unchanged. And second, the contents of the **Opcode, Operand X** and **Operand Y** registers are not written to the **Program Memory** until the **DEP+** button is pressed. Especially if only one word has to be modified, it is easy to make the mistake and switch to **PGM** mode, then find and modify the **Program Word**, and then hurry back to **RUN** mode to test it. Program memory is not modified if **DEP+** was not pressed after the modification was performed in the **Opcode, Operand X** and **Operand Y** fields.

If you use the analogy with Text Editor, **DEP+** command is similar to writing text in Overwrite mode, and **ALT / DEP+** in pushwrite mode.

Program (PGM) Mode: ALT / DEP+

If **ALT** is depressed, command **DEP +** not only writes the new word to the Program Memory, but also moves all the subsequent words in the Program Memory one place up, thus freeing one memory location. Then it duplicates the current word on the new location. There is no **UNDO**.

Here is the example:



Note: Command **ALT / DEP+** moves all subsequent program words one position up, which typically means that all symbol names should be modified. This has no effect only if the current instruction pointer is near the end of the written program and there are no symbol names behind it.

If you use the analogy with Text Editor, **ALT / DEP+** command is similar to writing text in Pushwrite mode, and **DEP+** in Overwrite mode.

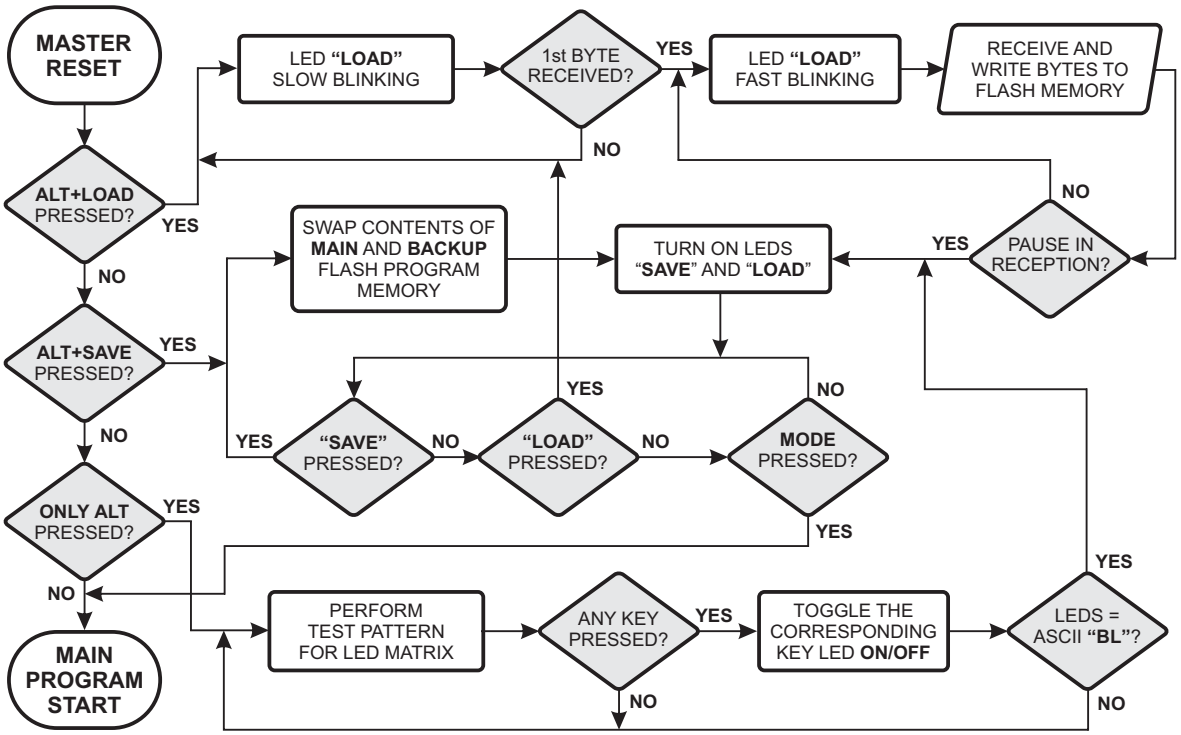
Special Modes

TEST MODE

Test mode is used for hardware testing. To enter **Test mode**, **ALT** button should be pressed at the moment when the unit starts operation after the hard **Reset**. There is no **Reset** button, but the **Reset** procedure may be performed by one of the two following methods:

- Disconnect and reconnect one of batteries, or
- Short pins **Ground** (third from the right) and **Reset** (the rightmost) on the I/O connector. Take care not to short pin **+3V** (which is between **Ground** and **Reset**) with **Ground**.

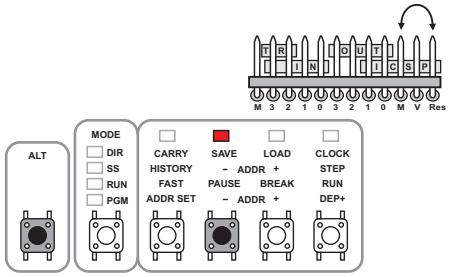
In **Test mode**, at the first moment all **LEDs** are in dynamic ON state, with the 1/3 duty cycle marching pattern. When the first button is pressed, all **LEDs** remain turned ON, except the display matrix and the lowest row, above buttons. Now you can toggle **On/Off** every **LED** individually, and thus test every button individually. When all the buttons are tested (all **LEDs** turned **ON**), the **TEST** mode is terminated and the unit returns to the normal operation.

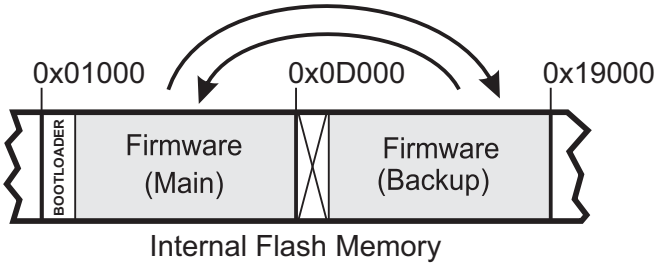


BOOTLOAD MODE

There are two functions in the **Bootload** mode: **SAVE**, which makes the backup copy of the current firmware in the extra space of internal Flash Memory, and **LOAD**, which downloads the new firmware from the computer, via **Serial Port**.

To make the backup copy of the current firmware, keep the buttons **ALT** and **SAVE** pressed during the **Reset** procedure (connecting the battery or shorting pins **Reset** and **Ground**). This swaps the **Main** and **Backup Program Memory** regions of the **MCU**, which takes about **5 sec**:

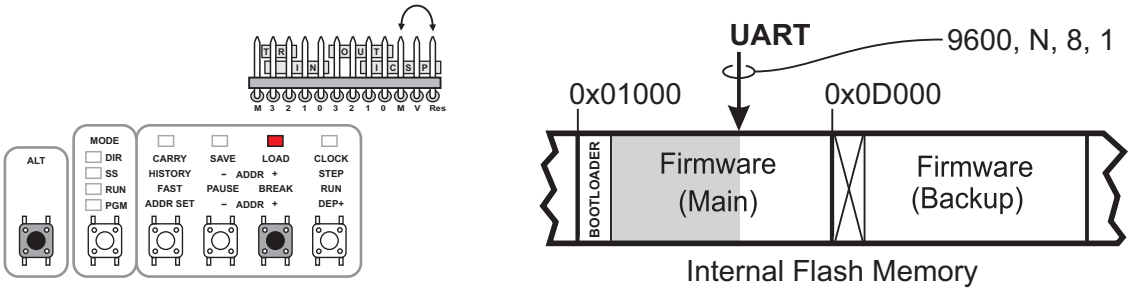




Special Modes

To download the new firmware immediatelly from the Serial Port, keep the buttons **ALT** and **LOAD** pressed during **Reset**. LED **LOAD** will blink slowly until the firmware starts loading, and than it will switch to fast blinking. This takes about **30 sec** or more, depending on the binary file length.

UART settings are **9600, N, 8, 1**. Settings in **SFR 0xF3 (SerCtrl)** do not affect the **Baud Rate** in **Bootload Mode** or for **Program Save/ Load**. Only pin **Rx** in the I/O connector is active, not the pin **Rx** on the **SAO** port. There is no transmitting from the unit, so although the pin **Tx** can be connected, it has no effect.



After any of the two procedures (**Swap** or **Bootload**), LEDs **SAVE** and **LOAD** are both turned **ON**, which means that the unit is still in **Bootload** mode. Only two buttons are active in this mode:

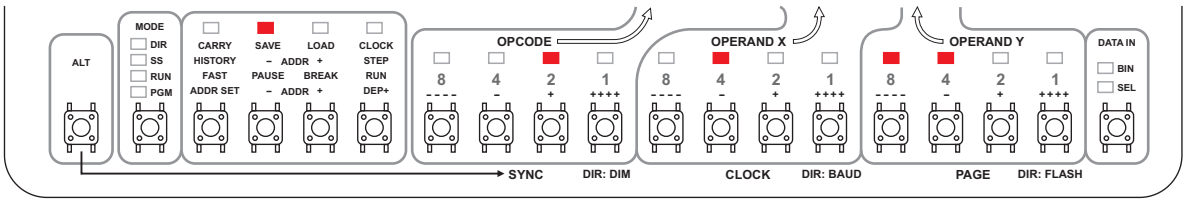
MODE: Return to normal mode
LOAD: Download the new firmware again from the **Serial Port**.

If there was no meaningful firmware in the **Backup** area and the **Program Memory Swap** is performed, command **MODE** has no sense, as the program will execute instructions from the unprogrammed area and will probably crash. In this case, **ALT-SAVE** with **RESET** should be performed again, to restore the old firmware, or **ALT-LOAD** during **RESET**, and the proper firmware downloading.

After the **bootload** process is complete, you can switch back to the normal mode, by pressing the button **MODE**. The firmware parameters **Version/Revision/Year/Month/Date** will be indicated at the top of the **LED** matrix, and the calculated **Checksum** at the bottom of the matrix. If the checksum matches the 16-bit number published together with the version, it means that the **bootload** process was **OK**. Between the **Version** and **Firmware Checksum** data (in rows **10** and **11**), there is the **Bootload Firmware Checksum**, in the same format as the **Main Firmware Checksum**. The **Bootload Firmware** area is write protected, so the checksum should not change in any circumstances.

Checksums are indicated in the **Big Endian** format, which means that the **High Byte** is displayed in the row **14 (0xE)** and the **Low Byte** in the row **15 (0xF)**. The Most Significant Bit (**MSB**) is at the left side. After the first button is pressed, **Version** and **Checksum** data will disappear from the LED matrix and the only way to see it again is to reset the unit again.

There is the alternate way to enter **Bootload** mode: while the unit is in Test mode (when the LEDs are blinking or all schematics LEDs are ON), you can test every button by switching the corresponding LEDs ON/OFF. You can use it to adjust **LEDs** in the **Command** and **Opcode** fields to display the **ASCII** uppercase **“B” (01000010)**, and also uppercase **“L” (01001100)** in the **Operand X** and **Operand Y** fields (**BL** is the abbreviation of **BootLoader**). Here is how it should look like:



When the **BL** “passcode” is recognized, the unit is automatically switched to the **Bootload** mode. In this mode, LEDs **SAVE** and **LOAD** are turned ON, and only buttons **MODE, SAVE** and **LOAD** are active.

Special Modes

SYNTHETIC INSTRUCTIONS

Synthetic Instructions or **Pseudoinstructions** are instructions which are not present in the regular instruction set, so they have to be synthesized using existing (naitive) instructions. Some of sytnetic instructions need more than one naitive instruction to synthesize the new one.

Non-existing instruction		Replace with	
RLC RX, RY	Rotate Left **RY** through Carry	**MOV RX,RY** **ADDC RX, RY**	Note: Result is in **RX**
SL RX, RY	Shift Left **RY**	**MOV RX,RY** **ADD RX, RY**	Note: Result is in **RX**
LSR RY	Logical Shift Right **RY**	**AND R0, 0xF** **RRC RY**	
CPL R0	Complement **R0**	**XOR R0, 0xF**	
CPL RX, RY	Complement **RY**, write to **RX**	**MOV RX, 0xF** **SUB RX, RY**	Note: Result is in **RX**
NEG RX, RY	Negate **RY**, write to **RX**	**MOV RX, 0** **SUB RX, RY**	Note: Result is in **RX**
NOP	No Operation	**MOV R0, R0**	Note: Any Register except **R12** and **R13**

Although Synthetic Instructions are generally the good replacement for some non-existing instructions, care should be taken about how they affect the flags.

AUTOMATIC OFF

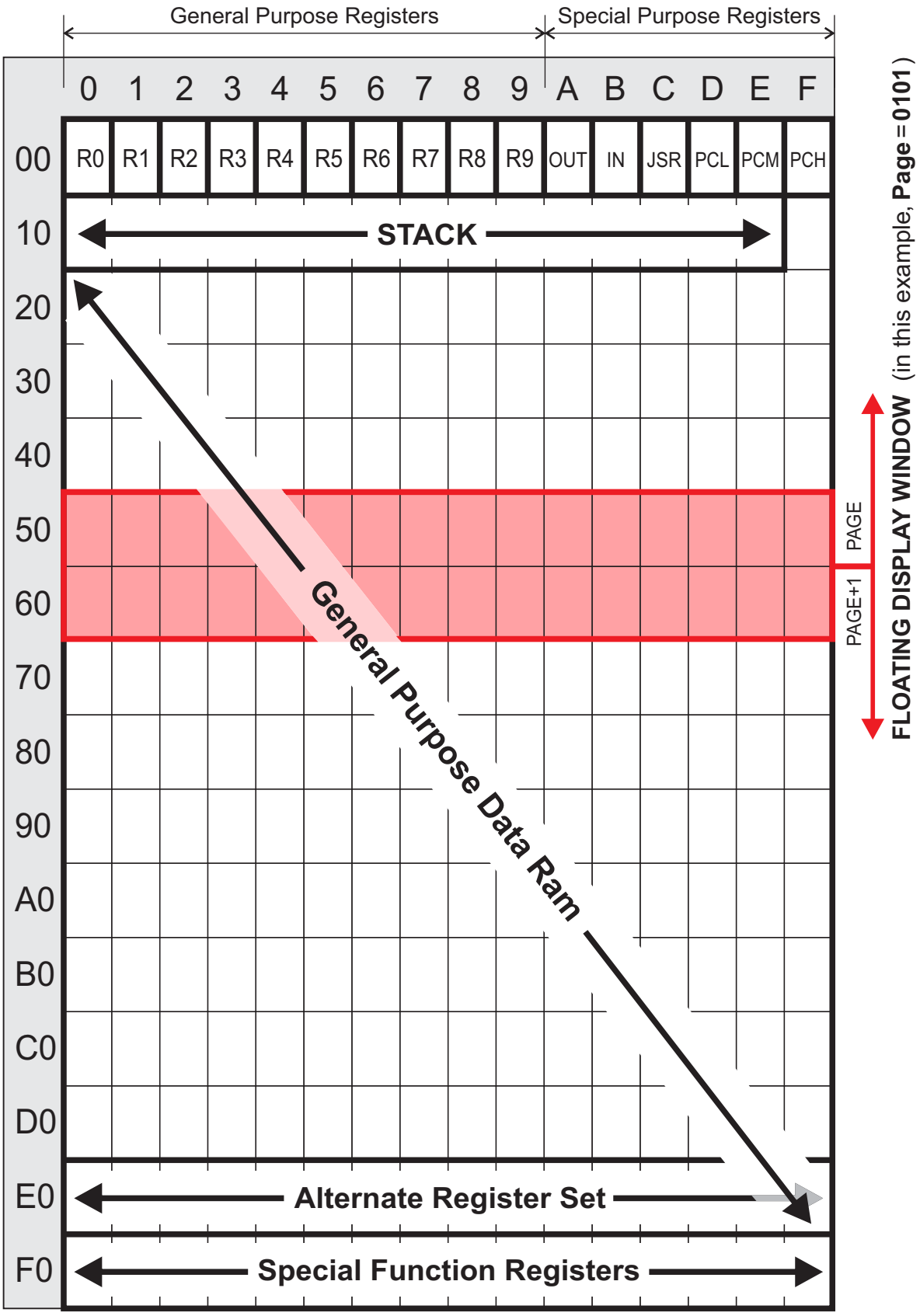
There is the internal countdown timer which counts independently of all other registers and, when it reaches zero, it turns the unit off, in the same way as the button command **OFF** should do. The upper nibble of the timer is in the **SFR** area, at the address **0xF9**, with the symbol name **AutoOff**.

Register **AutoOff** counts in the **10** minute resolution. It is readable and writeable, so when the processor writes a number in the **0-15** range, the unit will be turned of automatically in **10×N** minutes. Writing **0** to the register switches the unit momentarilly, and writing **15** will switch it off in **150** minutes (**2.5** hours).

At the Master Reset and **ON** command (button **ON-OFF** pressed while the unit is **OFF**), this register is preset to **2** (binary **0010**). That means that the unit will be switched **OFF** after **20** minutes if no other button is pressed in the meantime. At every button press (while the unit is **ON**), value **15** (binary **1111**) is written to **AutoOff** register, so it will be switched **OFF** after **2.5** hours of total inactivity.

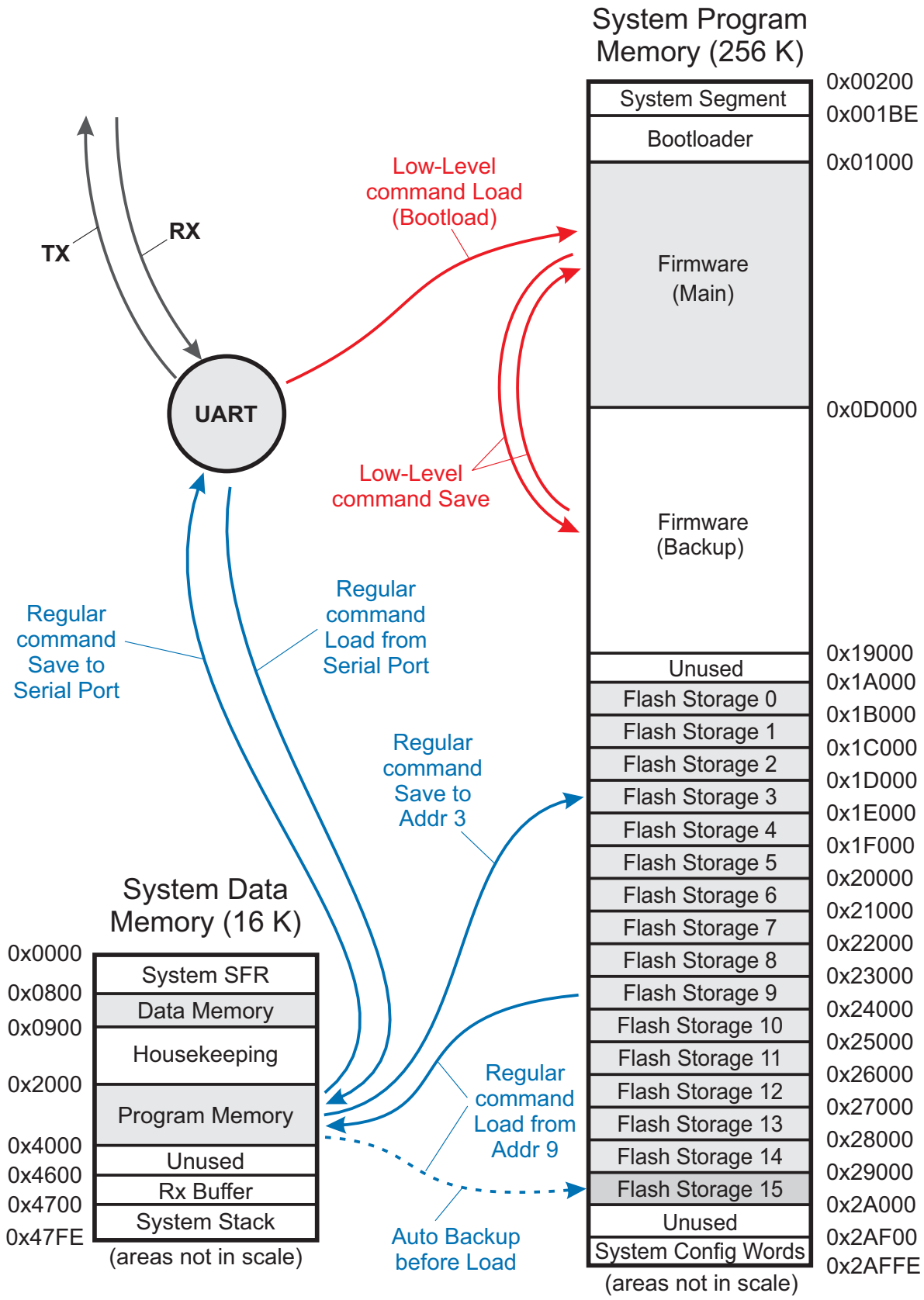
Sometimes it is required that the program works longer time without human supervision or intervention. The simple way to solve this problem is to put the instruction which writes **1111** or some other value in the register **AutoOff**, in the main program loop.

Data Memory Organization (simulated processor)



Note: this is the representation of the Data Memory for the hypothetical simulated processor, not the system processor PIC24FJ256GA704.

Memory Organization (system processor)



Note: this is the representation of the Program Memory for the system processor PIC24FJ256GA704, not the hypothetical simulated processor.