C H G - S O F T & M E D I A

NSC OPERATING SYSTEM

SOFTWARE DEVELOPMENT KIT

NO INTERPRETER VERSION 1.29

AppWizard version 1.1.40
DisplayMaker.dll 1.0.0.4

Table of Contents

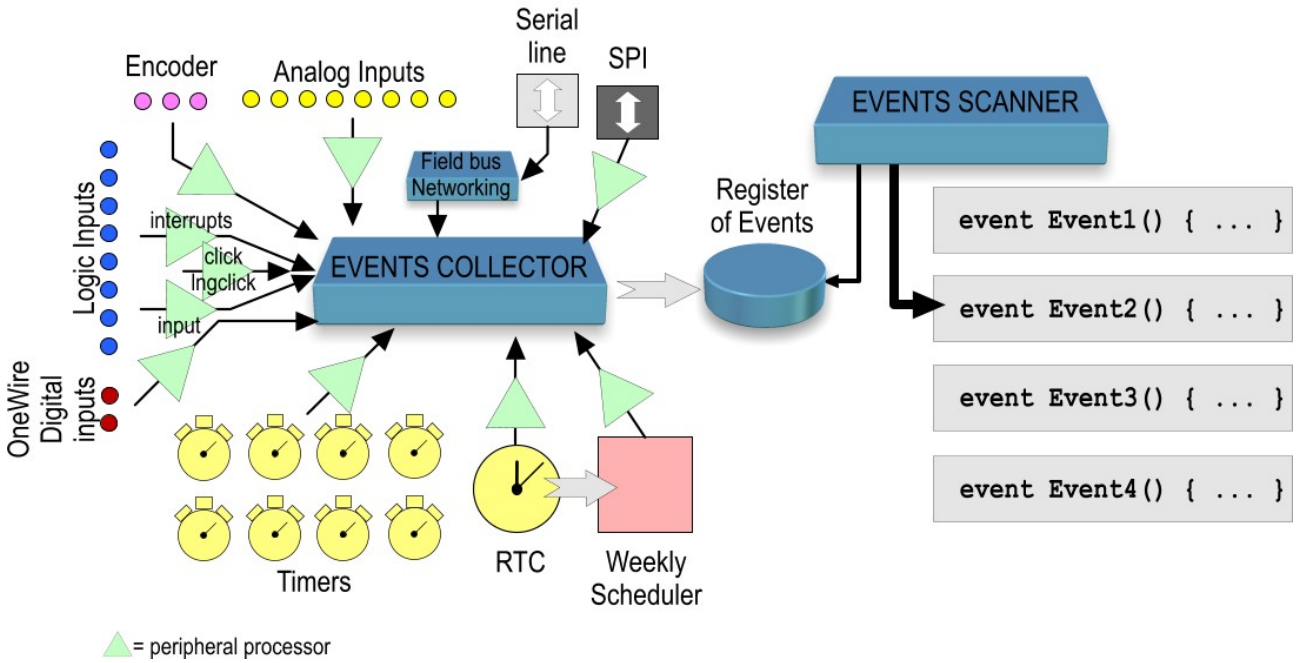
OVERVIEW

The NSC (Network Shared Control) operating system is designed to offer cooperative multitasking event driven environment for embedded processors, with SFBP based networking support.
The basic version is designed for the ATMEL Mega series, and in particular for the ATMega32 processors.

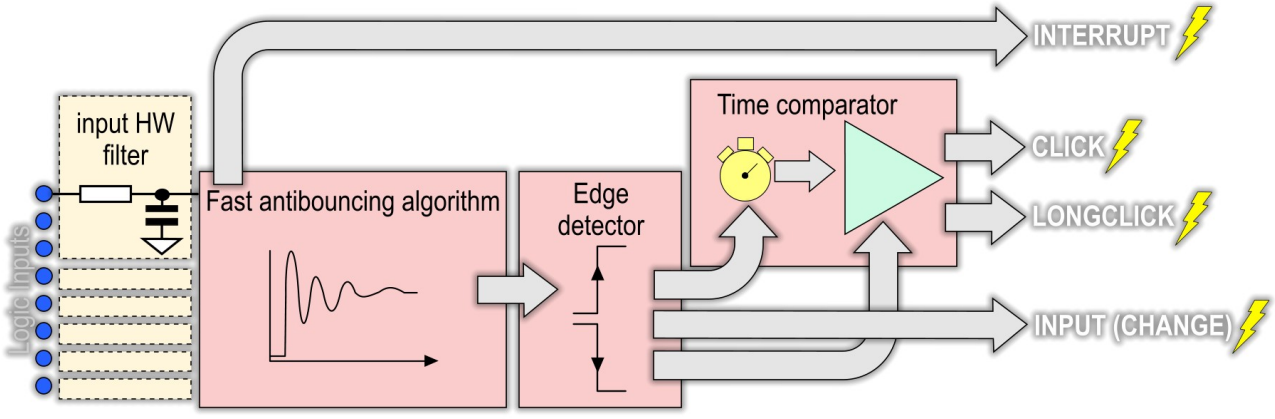
The OS features the following functions, some of them can be selected as optional to reduce the memory load.

- Event driven
- SFBP (Simple Field Bus Protocol) based networking with remote procedure call support
- Custom event
- Date and time with automatic DST
- Scheduler for logic or analog events
- Analog events
- Analog reading, with selectable number of analog inputs
- EEprom storage with simple read and write instructions
- Antibouncing for logic inputs, with integrated Input, Click and Longclick events
- Graphic, monochromatic display primitives (character based with some graphic capabilities)
- One wire reading for iButton and temperature sensors
- 8 timers
- Encoder A, B, Z counting algorithm
- Motor driver control
- Dimmer control
- Logic outputs including virtual outputs (non physically existing outputs)
- PWM output control

NSC system framework



IO Events subsystem

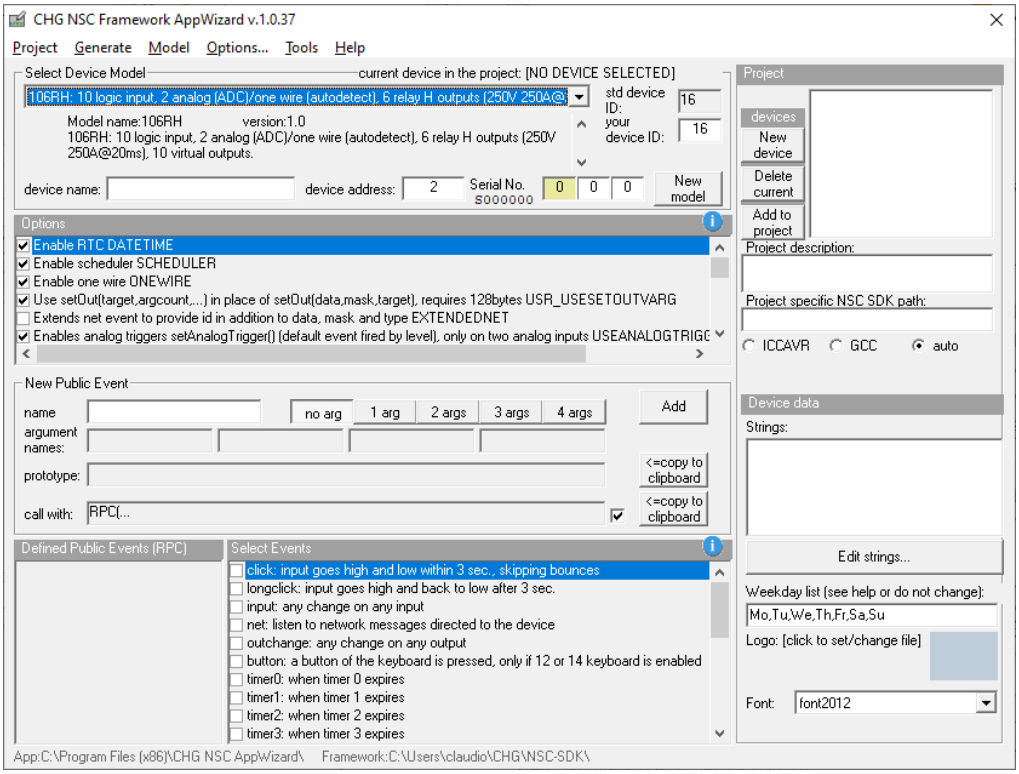


NSC FRAMEWORK

The NSC framework provides the O.S. and the framework to build on it your own application. The application is made of a main initialization section, functions, events, public events, and an optional loop control. All of them are detailed below.

AppWizard

The AppWizard is a desktop application that enables you to build your project, that can be composed by one or more objects or devices.



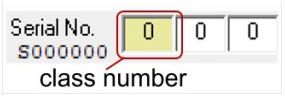
This tool easily allows to select the desired options and prepares for you the structures for public events and the related remote procedure call prototypes.

For devices equipped with a display it also allows to draw pages and chunks of text (Strings) to be used to render the desired information on screen.

USAGE

Add a device (object)

1. Select one device from the top list.
2. Give it a device name. Serial number and address are generated automatically, though you can change them. You can also enter a serial class number into the first box on the left. All numbers will be automatically generated starting from that serial number class.



Advanced: Optionally change the device ID. This is a rare necessity for special purposes.
3. On the right pane Project click the button Add to project. Now the new object is listed under the project list.

4. Make sure the object is selected ☑, then select the desired Options, select the desired Events .
5. You can also enter your own Public Events: enter the event name, select the number of arguments that the event can accept, and enter the related argument names. For more information see Pubevents. Once ready click Add to add the new public event to the list Defined Public Events (RPC) .

Add a new device

1. Click New device on Project on the right pane, follow the same steps listed above.

Create a new model

You may need to customize your own device model to suit your own hardware. To perform this click on Model and then Your own models... menu.
Into the Model editor that shows up, enter a proper device ID that is different from any other, add a model name, the hardware is based upon, and a descritpion.
Enter the device file name and path where you have stored the model's file. This should has **.c.h** extension. See the chapter ***Customize Your Device*** to create a file starting from a template.
Select the available capabilities the device can support and the options you want to make available when the model is selected in AppWizard.
Select what analog inputs are used (this is based on the ATmega32 or compatible processor).
Click Create/Save to finish or Cancel to exit.

Remark: The Model editor is mostly designed for ATmega32 or Mega series processors.

Modify a custom model

After you created a new model you may want to update or change or even delete it. Click on menu Model and then Your own models... to display the Model editor, and make the desired changes. When done click Create/Save to save changes, or Delete to remove the model altogether, or Cancel to just exit doing nothing.

Enter a custom logo

Where the device has a display a custom logo can show up at startup. To select the logo click on the logo box, and choose a proper file. Valid files are monochromatic BMP with size of 56x40 pixel exactly. Once an image has been loaded loading another picture and cancelling the dialog box will prompt a question whether you like or not to remove the logo.
The logo is common for the whole project.

Edit strings and pages

If the model device has a display you can create a collection of strings that will be used to draw pages or print texts on screen.
Click button Edit Strings... the ***Display Maker*** will show up (see below).

Specific path and compiler

If you are editing your project's files in a different computer where the actual NSC-SDK and compiler are located, you can enter the specific path into Project specific NSC-SDK path text box. You can also select for which compiler it is meant. Options are ICCAVR or GCC. Auto provides an automatic selection that normally is equivalent to selecting ICCAVR.
To access this configuration click on Project Specific Settings button.

Generating the project

Once done with all your devices in the project, options, strings, etc. you are ready to generate the project.
Click on menu Generate and choose where the files should go. You can choose among Windows or Linux style path, and whether the file should go into a single directory or into subfolders.

Generated Code Framework

The generated code is organized as follow.

Per each device up to ten files are generated. %%DEVNAME%% is a placeholder of each device name.

%%DEVNAME%%.c	This file, is the entry point of your program. Other files *.c.h and *.h will be included from here. The USERPROGRAM_HDR and USERPROGRAM_C respectively define the header and implementation files of your application. You will fill your code into these specific files.
%%DEVNAME%%.c.h	Where the code of your application lies.
%%DEVNAME%%.h	Header of the code of your application.
vectors.h	RPC vector file, this file is common for all the devices in a project. You can pick from here the name for the vectors to be used in RPCs.
Your own code will go here:	
%%DEVNAME%%.pvr.h	private header for your own global variables, defines, and prototype functions
%%DEVNAME%%.pvr-ini.c.h	your own iniziatization, run one time at start up code goes here
%%DEVNAME%%.pvr.c.h	implementation of your own functions
%%DEVNAME%%.evt.c.h	your implementation of events and pubevents (public events)
Other special files:	
%%DEVNAME%%.loop.c.h	Optional, if specified in Options this file can hold the code that must run into the main loop. Useful for polling functions.
%%DEVNAME%%.ms.c.h	Optional, if specified in Options this file can hold the code that is called by the system tick timer so it is processed every millisecond. Useful to run processes at regular intervals.

Special code before and after interrupts

For custom code to be executed before, and or after interrupts are enabled the additional files should be created including your code. Then uncomment the relative defines into %%DEVNAME%%.pvr.h :

- %%DEVNAME%%.inibfr.c.h file name for initialization code before interrupts are enabled, also uncomment #define INCLUDEINIT_BEFOREINTERRUPTS .
- %%DEVNAME%%.iniaft.c.h file name for code to be executed after interrupts are enabled, also uncomment #define INCLUDEINIT_AFTERINTERRUPTS .

Loop.c.h and ms.c.h

These two files are optionally used to include section of code that should run into the main loop and into the main internal timer respectively.
In both cases care must be taken to avoid jeopardizing the OS operations. In particular the millisecond code must be as short as possible.

Optional Private UART code

It is possible to manage directly the USART code for initialization, reception and transmission.

Private USART Initialization

In %%DEVNAME%%.pvr.h define PRIVATE_UART0_INIT
And in %%DEVNAME%%.pvr-ini.c.h write your initialization code.

Private USART receiving and transmitting code

In %%DEVNAME%%.pvr.c.h write your own function to receive and transmit data from USART.
You must also define PRIVATE_UART0RX_INT and PRIVATE_UART0TX_INT in %%DEVNAME%%.pvr.h .

Private RX and TX code to include into the standard communication module.

Save your RX and TX code into a file (one file for RX another for TX) and in %%DEVNAME%%.pvr.h add these defines:

```
#define INCLUDE_UART0RXINT_CODE          "yourRXfilename"
#define INCLUDE_UART0TXINT_CODE          "yourTXfilename"
```

Merge and initialization

After compiling, if it is required to merge the code with the bootloader and other optional components specific for the device, Blender.exe can be used for the purpose: just pick up a project for the same device as a template and change the code component and output file, eventually save as new project.
The file generated by Blender.exe is an Intel HEX format suitable to be uploaded into the target device.

If the application needs an initialized eeprom you may just use HexMaker.exe, enter the required data and location to store data into the eeprom, and generate the hex file.

You can then use either AVRdude or the AVR ISP to send the generated file into the target device's eeprom, as well as to send the final firmware in flash.

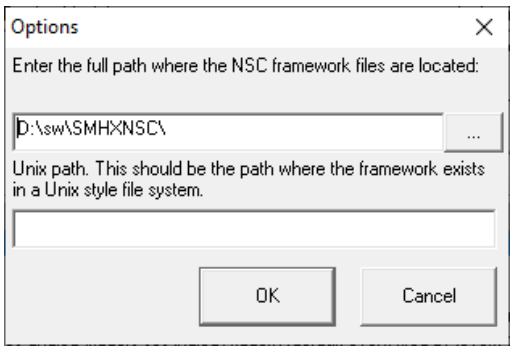
ICCAVR project configuration:

Open Project > Options...
Settings:

Target:	ATMega32 (or whatever the DEVICE requires)
Program type:	Application (bootloader: none) We actually add a bootloader by merging the specific bootloader with blender.exe but the application never start from the bootloader because it is reserved for firmware updates only.
PRINTF version:	small
String in flash only:	CHECKED
Return stack size:	64
Other options:	-bupfw:0x7010 -bwrjmp:0x7f00 The "other options" direct the compiler to reserve two symbols for jumping into the bootloader which is merged with blender.exe.

AppWizard configuration

Click menu Options... The full path to the NSC framework files should be found. Use the text box on top for Windows, and on bottom for Unix/Linux. The installer should already have filled the Windows path.



Display Maker

Display Maker allows to enter pages or texts to be displayed on the device's screen (where applicable).

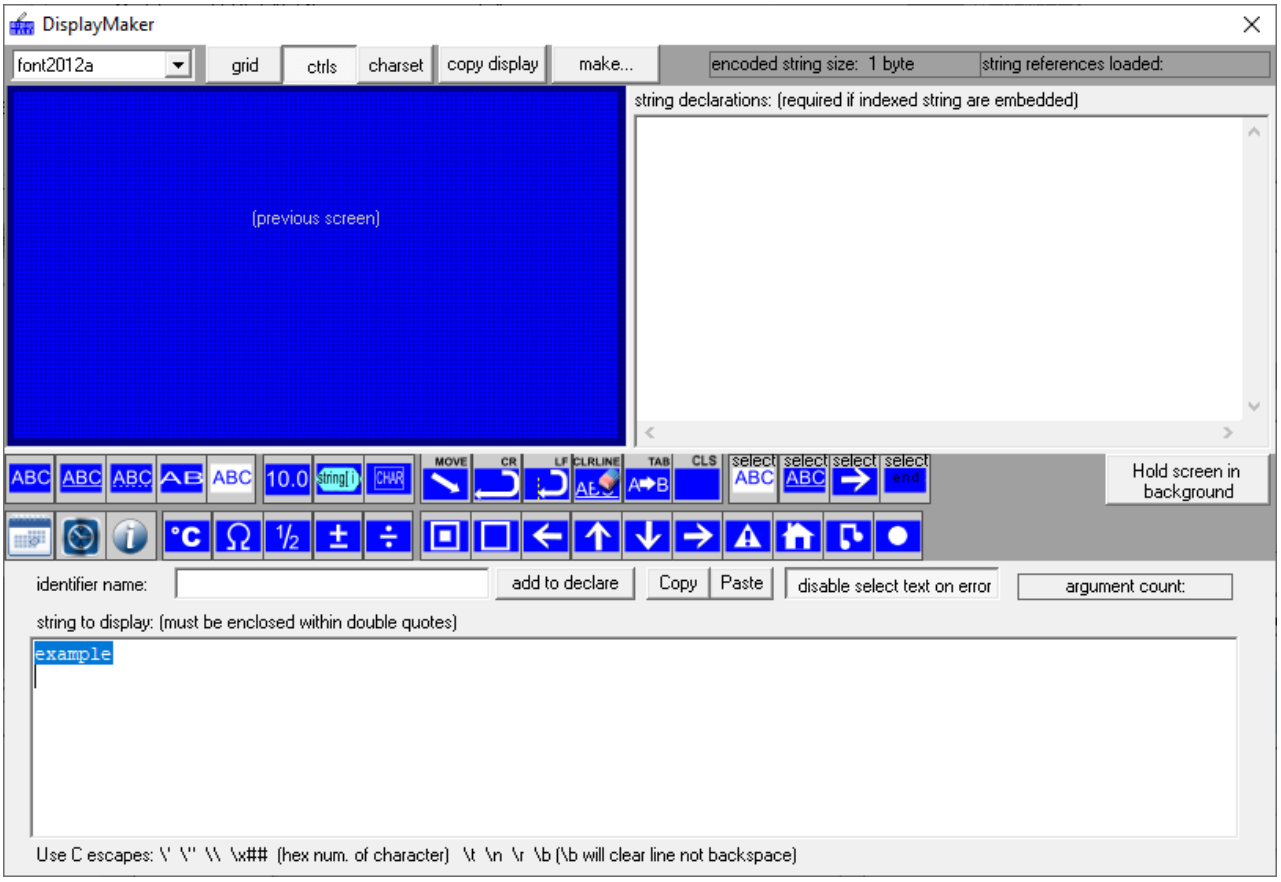


Figure 1: Display Maker

How to create a string

1. Enter an identifier name for your string, identifiers must have characters from a to z (upper and lowercase) only, it can also begins and ends with square brackets '[' ']' to group it into an index (see indexed strings below);
2. Into the text box at the bottom enter the string, this is the actual string command (see below);
3. Click add to declare to store the string, that will appear in the top-right pane of declarations.

It is not recommended to modify manually the strings into this pane, with the exception of deleting a whole row to remove a cut and paste a whole row to move it up or down.

Closing ***Display Maker*** will automatically update the list of the strings into the ***AppWizard***.

To edit a string already stored, click on "string ref. loaded " on the top-right corner (a hand cursor appears when hovering) to display the list of strings, double click on a item to edit the related string.

Entering or editing a string

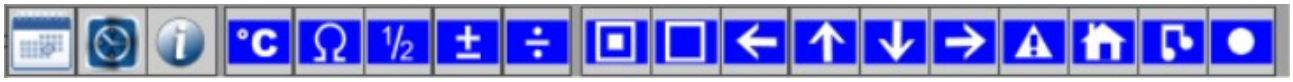
Type the string into the text box at the bottom. Strings not only contains text to display but also can contain commands that the printing function interpret to format text or for other special purposes.

For simple formatting you can use C escapes syntax `\r` for carriage return or  , `\t` for tab or  , etc. To enter a slash, a single or double quotation mark it needs to be prefixed with another slash, as in C.

`\b` or  clears the line and won't work as backspace, while `\n` or  advances to the new line aligning on the left at the last `move` left position.

Other commands:

normal text | underline | dotted underline | format large | inverted | format number variable | string ref. indexed | special character | move | CR | LF new line | clear line | Tab | clear screen | selectable invert | selectable underlined | selectable arrow | end selectable items



display date | display time | display version | special characters as shown

Formatting commands remains active as long as they are not cancelled by the command *normal text*  .

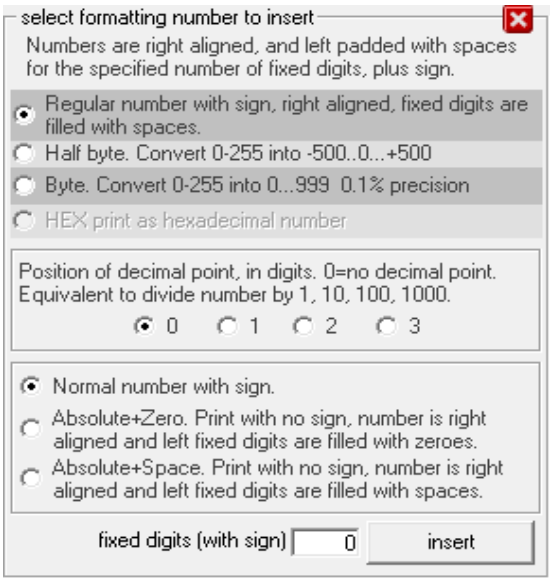
Move command

The command moves the cursor (or insertion point) at the position specified. Following new line commands will cause the text to advance by one line and move to left at the left position specified by the last move command.

Clearing lines

The command *clear line* clears the whole line from the point in which it appears (the characters before are unaffected). If a previous *invert*, *underline* or *dotted underline* formatting commands was issued the line is cleared accordingly with that format. Example "`\$invert;\b`" creates a white band.

Formatting numbers from variables



Number can be formatted in serveral ways as shown by the picture aside. Once inserted Display Maker put a numeric placeholder at the position where the number will be displayed.

You need to provide a number or a numeric variable as argument to `textOut` function. AppWizard add clues to help figuring out what argument is required when it generates the strings into the strings section.

Example, if two numeric variables have been inserted the function textOut is expected to receive those variables as the same order they are displayed.

```
textOut(     0, displaynum,
             <decimal num>, <byte to BCD>);
```

Selectable items

Buttons  allows to enter a command string to create a selection point that is automatically handled. First choose the desired style of selection clicking one of the first three buttons before each item, then after the last item click  to revert to normal.

On the left is showcased a three selection member example, on the right the related string.


```
\xd2Selection:\r
\xd5item 1\r
\xd5item 2\r
\xd5item 3\$normal;
```

Notice that going to a new line in this text is for reading purposes only, carriage returns are ignored and only the `\r` escape is recognized.

Selectable items can be highlighted printing the text and giving the index of the current item selected to the `textOut` function.

Example. The above example would generate this clue: "`textOut(0, Selects, <cur. selected item>);`" so in your code you would make the following call, giving 1 `<cur. selected item>` to highlight *item 2*:

```
textOut(0, Selects, 1);
```

with the result visible on the right:

Indexed strings and referenced embedded strings

You can create an array of strings that can be embedded into another string. The embedded string can be indicized so the displayed text will match the indicized string.

To create an array of strings surround the identifier name with brackets. You can enter multiple strings with different identifier names even non consecutively. All them will be gathered into a single array of strings.

Example.
Two strings are entered with identifier name 'one' and 'two':

```
[one] = "item one";
[two] = "item two";
```

Now create a string that embeds the first item of the indexed string, clicking on button  and selecting "one". The resulting string and related display will be like this: `displaysel = "Current: \#one;";`

To print this screen Display Maker hints to use the following: `textOut(0, displaysel, <index>);` where the argument `<index>` is any expression that returns a number between zero and the number of items in the group - 1. So you may call the function this way:

```
textOut(0, displaysel, index);
```

Remarks: Because you may have different groups of indexed strings gathered together into the same collection, it is important to enter the indexed strings in order, even though not necessarly consecutively. For example, you may have a group that lists the selected item, and following a group that lists the status:

```
[one] = "item one";
[two] = "item two";
[on] = "ON";
[off] = "OFF";
```

Then to show a string that displays the current status use something like:

```
dispstatus = "status: \#on;";
```

If <*index*> is a number that goes beyond the boundary limit of the array of strings textOut ignores the command and do not print the string to avoid a buffer overflow.
Caution: Indicized strings could contain formatted numbers or other indicized strings embedded, or selectable items. However you cannot embed an indicized string that embeds another string or has a formatted number or a selectable item as this would require nested arguments. When ***Display Maker*** detects such a condition a warning is issued, and you should remove the reference to the indicized string.

Overlapping texts and Pages

To create a page the first entry should be a *CLS* command to clear the screen and reset all formatting commands. This ensures the following text is printed on a clean background.

However you can also print overlapped texts. This is useful in several cases, for example to display a menu on top of the current screen. Or to update an indexed string.
To place an overlapped string on the correct location on screen consider starting the string with a *move* command. This moves the string without affecting the background.
While you can also use *Tab* or *CR* for this purpose they wouldn't reset the horizontal and vertical positions though.

Tip: For indexed strings that need to be printed on the same target location as referenced embedded strings into a page, it could be useful to make all them of the same size, padding with spaces if necessary.

Hold screen in background

When creating overlapped texts you can find useful to have view how the page would behave. For this purpose first select the page which other strings would be printed on top, then press Hold screen in background button. Now swicth to the overlapping string to edit. The page is retained in background so you can have a better view where the text will go and the effect of the various commands.
Remark: For simple visual purposes the held screen also reset the X-Y coordinates, but in a real situation this couldn't be the case. Always consider to start a "transparent" overlapping string with a *move* command.

How to display an overlapped menu

Create a string for the base page, make sure to start with *CLS*. Then make a new page with no *CLS* (that is a "transparent" page) but with a *move* command to the location where the menu should be printed. This last page could have selectable items in it. In such a case it is advisable to make the item of the same size, padding them with spaces if necessary.

On code:
1. print the background page.
2. print the overlapping menu page.
If the selected item changes, repeat printing the menu page this time giving the new selected item as argument of `textOut` (see above, Selectable items).
3. When done, to hide menu, print again the background page.

How to display an overlapped number (or numbers)

Suppose an application where some values need to be updated on screen.
You have two options:
a) Create the page with the number.
b) Create the background page, then create a string with the sole number (and *move* commands).

On code:

case a):	Print repeatedly the page with the updated numbers.
case b):	1. Print the background page;
	2. Print repeatedly the string (the "transparent page") with the sole updated numbers.

In (b) case the move command is required to position each time the number to print at the right location on screen. Be sure to have padded numbers or preceed the number with a *clear line* or spaces followed by a new *move* back.
Depending on the situation the option (b) could be faster and with less flickering, in particular if a couple of number need to be printed. If multiple number need to be updated then the solution (a) could be favorable.

Back to AppWizard

Once done with the strings you can go back to the AppWizard simply closing Display Maker: the strings are automatically stored in AppWizard.

GENERAL

GLOBAL CONSTANTS

`this`	The same device/object. It can be used everywhere a `target` or `address` or `object` argument is required.
`false`	0
`true`	1
`ON`	1
`OFF`	0

Version info: `VERSION_MAJOR` `VERSION_MINOR`

```
DEBUGSEND(I1,I2,B1,ID)
```

Send debug information to the station address (1) via SFBP.

I1, I2	integer data
B1	byte data
ID	must be a number between 0 and 30, it is send added to CTRLTYPE_USER

```
_tick
```

Global circular tick counter which is pseudo-random initialized. It is continuosly increased by the base-time timer 0 interrupt so that it may be used for timing difference or as absolute number for random numbers. Range: 0 - 255.

SYSTEM EEPROM MEMORY LOCATIONS

`MEM_TIMEBASE`	Time storage. **From this location and above no user data should be stored or retrived.**
`MEM_ERRBASE`	Base location where error data is stored
`MEM_LOCADDR`	Local address

GLOBAL TYPES

```
typedef     unsigned char     byte;
typedef     unsigned char     bool;
typedef     void              event;
typedef     void              pubevent;
typedef     signed short      VARVALUE;
typedef     unsigned short    UINTVAR;
->    typedef signed short EncoderValue_t;
->    typedef long EncoderValue_t;              if LONGENCODER is defined
```

IO TESTING

Test a single bit (single IO by providing its number):

```
TESTIO(iodata, ionum)
```

`iodata`	any IO buffer data provided by the event Input, Click, Longclick, Outchange
`ionum`	number of the input to check. Inputs are enumerated starting by zero.

Macro operation: `((iodata) & (1 << (ionum)))`

Test multiple bits (multiple IOs by providing their bits), this macro returns true if ALL the required IOs are set. Caution: This does not work when the IOs to check must not be set.

```
TESTIOs(iodata, ios)
```

`iodata`	any IO buffer data provided by the event Input, Click, Longclick, Outchange
`ios`	group of bits each one representing one I/O, where bit zero is I/O zero

Macro operation: `(((iodata) & (ios)) == (ios))`

KEYBOARD TEST MACROS

When a 4 wires keyboard is connected to the inputs the following macros can be used in conjunction with events click, longclick and input to check what keys has been pressed.
A 4 keys keyboard directly drives the inputs, so it is possible to check for combination of pressed keys. On a 6 to 15 keys keyboard no combination of keys can be read.
A 4 wires keyboard should be connected to inputs 4, 5, 6, and 7.

4 wires constants

```
IO_KEYDW          0x0010      this is the v button in a 4 buttons keyboard
IO_KEYUP          0x0020      this is the ^ button in a 4 buttons keyboard
IO_KEYENTER       0x0040      this is the O button in a 4 buttons keyboard
IO_KEYCANCEL      0x0080      this is the X button in a 4 buttons keyboard
IO_KEYALL         0x00F0      any key
```

4 Keys Test Macros

`ISKEYENTER(x)`	Return NZ if ● is pressed
`ISKEYUP(x)`	Return NZ if ▲ is pressed
`ISKEYDOWN(x)`	Return NZ if ▼ is pressed
`ISKEYCANCEL(x)`	Return NZ if ✖ is pressed
`ISANYKEYPRESSED(x)`	Return NZ if any key is pressed.

NZ = non-zero.

6 to 15 keys Test Macros

`ISKEYCANCELD(x)`	Return NZ if Cancel is pressed
`ISKEYENTERD(x)`	Return NZ if ENTER is pressed
`ISKEYNEXTD(x)`	Return NZ if NEXT is pressed
`ISKEYPREVD(x)`	Return NZ if PREV is pressed
`ISKEYUPD(x)`	Return NZ if UP is pressed
`ISKEYDWD(x)`	Return NZ if DW is pressed
`ISKEYALTENTERD(x)`	Return NZ if ALT is pressed
`ISKEYF1D(x)`	Return NZ if F1 is pressed
`ISKEYF2D(x)`	Return NZ if F2 is pressed
`ISKEYF3D(x)`	Return NZ if F3 is pressed
`ISKEYF4D(x)`	Return NZ if F4 is pressed
`ISKEYF5D(x)`	Return NZ if F5 is pressed
`ISKEYF6D(x)`	Return NZ if F6 is pressed
`ISKEYF7D(x)`	Return NZ if F7 is pressed
`ISKEYF8D(x)`	Return NZ if F8 is pressed
`ISKEYHOMED(x)`	Return NZ if HOME is pressed
`ISANYKEYPRESSED(x)`	Return NZ if any key is pressed.

Usage example

```
event click(int Clicked)
{
	if(ISANYKEYPRESSED(Clicked))
	{
		// a key has been pressed
	}

	// suppose this is a 16 keys keyboard
	if(ISKEYF1D(Clicked)) {
		// key F1 has been pressed
	}
}
```

Numerical 15 keys keyboard

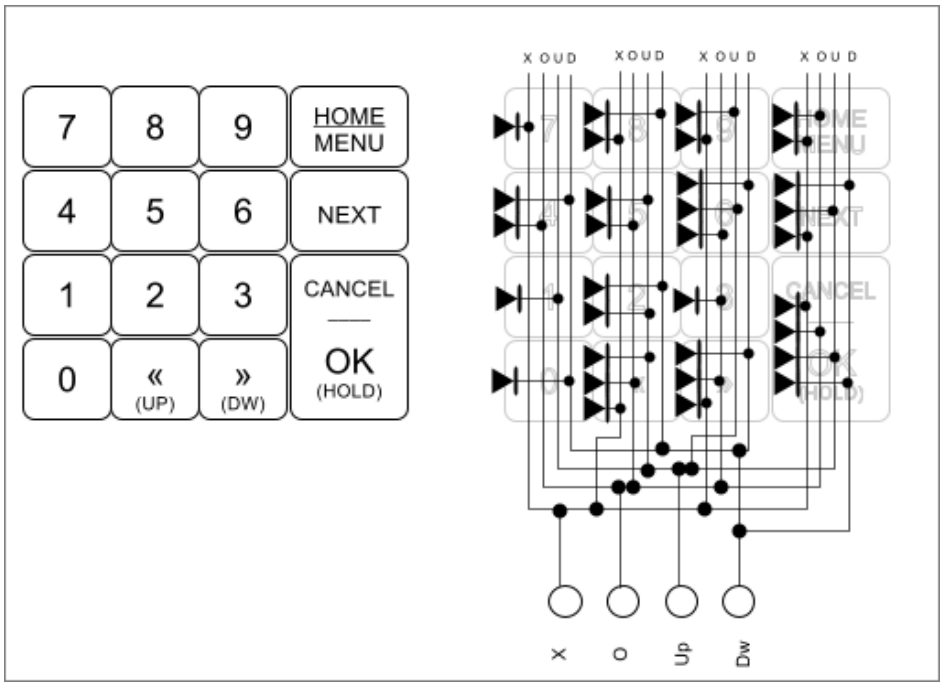
Numerical 15 keys keyboard (see also drawing)		Line (U=up, D=down, O=circle/enter, X=cross/cancel)			
Macro	*Key name*	X	O	U	D
ISKEYN0(Clicked)	#0				1
ISKEYN1(Clicked)	#1			1	
ISKEYN2(Clicked)	#2			1	1
ISKEYN3(Clicked)	#3		1		
ISKEYN4(Clicked)	#4		1		1
ISKEYN5(Clicked)	#5		1	1	
ISKEYN6(Clicked)	#6		1	1	1
ISKEYN7(Clicked)	#7	1			
ISKEYN8(Clicked)	#8	1			1
ISKEYN9(Clicked)	#9	1		1	
ISKEYN_NEXT(Clicked)	NEXT	1		1	1
ISKEYN_MENU_HOME(Clicked)	MENU (PREV) / HOME (longclick)	1	1		
ISKEYN_DW(Clicked)	>> (DW)	1	1		1
ISKEYN_UP(Clicked)	<< (UP)	1	1	1	
ISKEYN_CANCEL_ENTER(Clicked)	CANCEL / ENTER (longclick)	1	1	1	1

ISKEYNUMBER macro allows to check if the pressed key is a number (0 to 9).
KEYNUMBER macro allows to get the pressed numeric key, example:

```
unsigned char num;

if(ISKEYNUMBER(Clicked)) {
    num = KEYNUMBER(Clicked); // num now contains the pressed number
    putnum((PRNTVALUE)num, 0);// print the number (if device has display only)
}
```

Layout and electrical connections for a 15 keys numeric keyboard.



API - FUNCTIONS

GENERAL

```
int rnd(void)
```

Retruns a pseudo-random value.

```
void pwrsavebypass(unsigned short mask)
```

Set a mask of outputs that will not be saved when power save is invoked. Available with `USEPWRSAVEBYPASS` only.

```
void _setTComp(signed char T)
```

Set temperature compensation. `T` must be between -128 and +127. Available with `ONEWIRE` only.

```
void glowlamp(void)
```

Lits up the display lamp. The lamp is automatically turned off after a while. Available with `GLOWDISPLAYLAMP` display only.

```
void write(VARVALUE value, int location)
```

Writes `value` in EEprom at `location`, that can be zero to `MAXMEM`-1.
Also, `location` determines if the value must be written as byte or integer: if location has the most significative bit set then the value is deemed as integer. Alternatively `location` can be combined with `_RW_INT_` to request writing an integer.
Data are stored from `MEMSTORE`.

```
VARVALUE read(int location)
```

Retrieve the value stored at `location`. This argument determines whether to retrieve an integer or a byte: if the most significative bit is set in `location` then the value to read is deemed to be an integer. This can be achieved by combining `_RW_INT_` with `location` .

```
_DoTasks()
```

Yield control to task manager. This can be called if a function takes longer time to complete. However caution should be used to not overwhelm the hardware stack and not to cause deadlock or race conditions. In general it is not recommended to use this function!

Stack

```
void push(int data)
int  pop(void)
```

Respectively push to and pop data from stack. These functions are available only if MAXSTACK is defined. There is no way to know if the stack is full, or popped data is from an empty stack (in which case pop returns zero).
In almost all cases MAXSTACK is never defined.

```
void ResetAll(void)
```

Broadcast a CTRLTYPE_CLEARERR and CTRLTYPE_REBOOT message to all units, and then clear any error and reboot itself.

```
GLOWALRMLED;
DARKALARMLED;
```

Respectively lit up and turn off the alarm LED.

IO FUNCTIONS

```
int getIn(char channel)
```

Retrieve the status of the given input channel.

Logic inputs
- If the given `channel` is `LOGIC` or `this` or the local device's address, then `getIn` returns an array of 16 bits of the status of each logic input. Inputs that do not physically exist are set to zero.
- If the given `channel` is the address of a remote device, then the function returns an array of 16 bits of the status of each logic input of the remote unit.
You can use `TESTIO` or `TESTIOs` to check for specific inputs.

Analog inputs (Only when `HASANALOGICINPUT` is defined)
If `channel` is
- `ANALOG`*n* where n is 0 or 1 for devices with up to 2 analog inputs;
- `MAN`*n* where n is 0 to 7 for devices with more than 2 analog inputs, up to `ANALOGCODECHANNELMAX`;
then the function returns the value of the given analog inputs as a 16 bit integer.
`NUMCHANNELS` tells number of analog channel available. See also Reading ADC below.

Digital inputs (Only when `DIGITALCODECHANNEL` is defined)
If channel is
- `DGTDATA`*n* where n is 0 to 4 (or less than `DIGITALCODECHANNELMAX`) for devices with up to 4 digital inputs;
- `MDD`*n* where n is 0 to 7 (or less than `DIGITALCODECHANNELMAX`) for devices more than 4 digital inputs;
then the function returns a 16 bit value of the digital (one wire protocol, i.e. temperature sensor) input. You can also use `DGTDATA` + *n* instead of `MDD`*n*.

Remarks: It is possible to retrieve only logic inputs from a remote device. This is possible even if the local device don't has inputs at all. If the remote unit fails to reply a `_ERRRPCFAILED` error is issued and it can be handled using ***`try`*** and ***`catch`*** (see Error handling).

Reading ADC

By default `getIn` returns analog inputs with 8 bit resolution. Defining `ANALOG10BITS` it is possible to retrieve inputs at 10 bit resolution.
Normally ADC is read and compared with triggers and 24V brown out detection at a rate of 1 millisecond per channel. Defining `FASTANALOG` moves the whole process into the main loop, making each channel to be scanned at a rate of about 85µs with 8 bits resolution, and 90µs with 10 bits resolution.

Asynchronous ADC readings

Faster asynchronous reading can be achieved defining `FASTANALOG` and `ANALOGINTERRUPT` (both defined). This makes the ADC reading asynchronous (through interrupt). However `ANALOG`*x* events (if only 2 channels are used) are fired within the main loop at a rate of about 20µs.
For faster reading use `getIn` within file *%%DEVNAME%%.loop.c.h* (where *%%DEVNAME%%* is the name of your device) and make sure `LOOPINCLUDE` is defined. `INCLOOPTASKS` can be defined as well to have

this loop inside the Task manager wich make the operations working even while some other tasks are running.
An alternative is the use of a timer and put the reading routine inside its event.
Asynchronous ADC reading is incompatible with BRWN24 (24V brownout detection).

10 bits resolution

To increase resolution define ANALOG10BITS, this also requires FASTANALOG and ANALOGINTERRUPT defined as well. Use the function: `convert10Bit2Decimal` to get a decimal representation of the ADC data (see Graphic Library for details). Up to 8 analog channels can be read with this option.

12 bits ADC readings

Defining OVERSAMPLE_DECIMATION allows to add an Oversampling and Decimation algorithm to virtually increasing the ADC resolution at 12 bits. With this option all ANALOG10BITS, FASTANALOG and ANALOGINTERRUPT must be defined as well.
The increase in resolution is at expenses of speed that decreases by 16 times. So at best the updated value for a given channel is possible after about 400µs to 1.4 millisecond.
Up to 8 analog channels can be read with this option. Use the function: `convert10Bit2Decimal` to get a decimal representation of the ADC data, you also need HIPRECISION_HIRES_CONVERSION defined (see Graphic Library for details).

Averaging

With OVERSAMPLE_DECIMATION it is also possible to have a mobile averaging by defining AVERAGEADC. Average is spread over a cycle of 16 steps.

Direct ADC reading

Calling the function `getIn` could load some overhead. To avoid this use the following macro to directly read ADC from the internal buffer: `READANALOG(index)` where index must be an unsigned char with a value comprised between 0 and NUMCHANNELS-1 .
This macro returns the correct value at 8, 10 or 12 bits depending by the selected options.

Direct private analog scan

If PRIVATEANALOGSCAN is defined none of the ADC operations described above are performed/available. No events are triggered, nor data is read from ADC, including the 24V brownout detection. It is your own responsibility to create your procedure to scan analog inputs where you deem better for your application if this option is defined.

ADC conversion speed

Default (8 bit)	8Ksps, full conversion: 124µs
FASTANALOG (8 bit, 10 bit and oversamp&dec)	16Ksps, full conversion: 54µs
FASTANALOG + ADC500KHZ (8 bit)	35Ksps, full conversion: 108µs

MACRO

```
GETDS1820TEMPERATURE(channel)
```

Get the DS1820 sensor temperature for the given `channel`. About channel it can be 0 to digital max channel. The returned value is already converted into tenth of degree celsius, i.e.: 215 means 21.5°C.

```
unsigned int OUT(byte IOnum)
```

Returns the status of the given `IOnum`, which must be in range between 0 and 15. If the given `IOnum` does not physically exist the function returns its virtual value.
If `IOnum` is 16 or above the function returns the whole array of bits with the status of all 16 outputs (whether they are physical or virtual). You can then use `TESTIOs` to check for multiple outputs on a single operation.

```
int getOut(byte Object)
```

Retrieve an array of all 16 outputs, whether they are physical or virtual, of the given `object`.
`object` can be:
- `LOGIC`, `this` or the local device's address: the function returns the local outputs.
- the address or device/object name of a remote device: the function returns the remote outputs.

You can use `TESTIO` or `TESTIOs` to check for specific outputs.

Remark: If the remote unit fails to reply a `_ERRRPCFAILED` error is issued and it can be handled using ***try*** and ***catch*** (see Error handling).

```
byte setOut(byte target, byte argcount, ...)                (see note)
byte setOut(int data, int mask, char target)
```

Set one or more outputs.
Note: This function comes in two flavours: the first one is when USR_USESETOUTVARG option is selected, while the second one is when that option is unselected.

target	destination device, it can be this, the device's local address or the address or a remote device name.
argcount	number of following arguments (see below). (USR_USESETOUTVARG only)
...	(USR_USESETOUTVARG only) variable number of arguments holding the desired outputs to set. Use ON*n* or OFF*n* respectively to set the output on or off, where n is the output number in range between 0 and 15. Example: setOut(this, 2, ON0, OFF3);

The following apply if USR_USESETOUTVARG is not selected only:

data	array of 16 bits that defines the required output status: 0 off, 1 on. Example: 0b101 set output 0 on, output 1 off, output 2 on.
mask	array of 16 bits that defines the outputs that should be affected by the operation. Example: 0b11 makes only outputs 0 and 1 to change, the remainder will be unaffected by the operation.

Remarks: When USR_USESETOUTVARG is selected _setOut(data, mask, target) function is still made available and can be used in the case of special needs.

Examples:

```
      _setOut( IO2, IO0 | IO2, this);
```

Set IO0 off and IO2 on. Missing IOs on first argument are intended to be off.

```
      #define PUMP      IO0
      #define LIGHT     IO1
      int outON;  // hold outputs to set ON
      outON = PUMP;
      if(TESTIO(getIn(this), IN0)) outON |= LIGHT;
      _setOut( outON, PUMP | LIGHT, this);
```

Defines PUMP and LIGHT outputs, set on PUMP and LIGHT if IN0 input is on as well. Then set the intended outputs.

OUTPUT MACROS

the following macros call the underlying function ***_setOut***.

setPWM(PWMchannel,value)

Set PWM channel to the given value. Value must be an integer within 0 to 1000 to change modulation between 0 to 100% with a granularity of 0.1%.
PWMchannel can be either PWM_A or PWM_B.

Note: If this function is used when SERVO is selected as well, then only channel PWM_B should be used, and the accepted value range changes from 0 to 1000 into 0 to 1999.

servo(speed,current)

Set speed and current limit if the device supports servodriver. When SERVO is selected the output PWM_A is used, while PWM_B can be used with setPWM for other purposes.
With SERVO a tachometer feedback is expected at analog input *AN0* (pin PA0 on ATmega32). The voltage is normalized at 5V for the maximum speed. Analog input *AN1* (pin PA1 on ATmega32) is expected to receive the current in a normalized range of 0 - 5V.

Parameters:

speed	integer value between 0 to 1999
current	current cutoff limit set in range of 0 to 255 (max current)

Remark: When SERVO is selected the value to give to setPWM for the free channel PWM_B changes from 0 to 1000 into 0 to 1999.

dimmer(value,dimmerChannel,target)

Set dimmer (phase fire control PFC, or phase partialization control), if the device support PFC.

Parameters:

value	value in range between 0 and 20 (full on), or 40 for hi-res dimmer.
dimmerChannel	channel to control, it can be DIMMECH*n* where n can be 0 to 3, anyway up to DIMMERNUMCHANNELS. You can also use BASEDIMMECH + n .
target	destination device, it can be either this or the device's local address or the address of a remote unit.

ANALOG TRIGGER

```
void setAnalogTrigger(char lowerLevel, char higherLevel, char channel)
```

Set an analog trigger that will make the event `Analog`*n* (where n can be 0 or 1) to be fired if the analog input goes beyond the set levels.

`lowerLevel` lower threshold level below which the event is fired.
`higherLevel` higher threshold level above which the event is fired.

Remark: Available only if **`USEANALOGTRIGGERS`** is selected. Higher and lower levels set the hysteresis: the event will be fired if the level goes below lowerLevel if it went above the higherLevel, and above higherLevel if it went below the lowerLevel. Calling again `setAnalogTrigger` resets the event so it would be fired even if it didn't trespassed the opposite threshold.
Not available if `NUMCHANNELS` is greater than 2 and if `ANALOG10BITS` is defined.

ANALOG LEVEL

If in place of `USEANALOGTRIGGERS` is selected **`ANALOGLEVEL`** then an automatic event is fired each time one of the analog inputs change. To reduce the sensitivity `ANALOGSENSITIVITY` can be defined and `ANALOGSENSITIVITYVALUE` defined to a value comprised between 0 (lowest sensitivity that ignores the lower four bits) and 4 (maximum sensitivity: no bits are ignored).
Example: `#define ANALOGSENSITIVITYVALUE  2`

Remark: This option takes one byte of RAM per channel, an additional byte per channel is taken if `ANALOG10BITS` is defined as well.

ENCODER

```
EncoderValue_t encoder(char reset)
```

Returns the encoder counter's value, and optionally resets the counter. See also *`zeroencoder`* event.

`reset` `ENCREAD` the encoder counter's value is returned, no further action is taken.
`RESETONZERO` the counter value returned and sets the zeroencoder trigger. After this as soon as the zero pulse happens the *`zeroencoder`* event will be fired.
`RESETNOW` both zeroencoder trigger and counter are cleared.

Return: the value of the internal counter, this could be a signed short integer, or a long integer if `LONGENCODER` is defined.

Remarks: Available only if `HASENCODER` is selected. If `LONGENCODER` is selected then the returned value is a signed long integer.

Encoder limitations

Because of the speed limits of the processor the maximum readable frequency is capped at 5KHz for a three channel A-B-Z encoder.

TIMING FUNCTIONS

```
void setTimer(unsigned short interval, byte timerID)
```

Set timed interval. The timer runs independently if not cancelled by calling `setTimer` with `interval` set to zero. When the timer expires a `timer`*n* event is fired (where n is 0 to 7).

`interval`	number of tenth of seconds, up to 65535 (1h 49' 13".5) the timer will expire. If `CENTISECOND` option is defined then the timer expires 10 times faster.
`timerID`	ID of the timer, this must be `TIMER`*n* where n is between 0 and 7.

Remarks: Calling `setTimer` with zero interval cancels any pending timer of the same `timerID`. To get proper interval you can used macro `makeTimerInterval(h, m, s, t)` where **`h`** can be zero or 1, **`m`** can be 0 to 59, **`s`** can be 0 to 59 and **`t`** can be 0 to 9. Maximum is `(1, 49, 13, 5)` .
Becasue this is a macro, given values are not checked and you should ensure they are within the limits or a variable overflow may occur.
If `CENTISECOND` is defined then the interval is in cents of second. Also `NUMCENT_TIMERS` should be set to the number of cent timers required, that must be less or equal to all 8 available timers.
With `CENTISECOND` the maximum time is 10':55".35 you can use `makeTimerCentInterval(m, s, c)` to built a suitable value giving the interval in minutes (m), seconds (s) and cents of second (c).

```
unsigned short getTimer(byte timerID)
```

Get the remaining time of the given timer `timerID`.
If `DATETIME` is selected current time can be retrieved as follow:

`GET_SECOND`
`GET_MINUTE`
`GET_HOUR`

`GET_DAY_WEEK` returns a combined day and weekday, use `FILTERDAY` to extract the day and `FILTERWEEK` to extract the weekday.
Example:

```
byte myday = FILTERDAY(getTimer(GET_DAY_WEEK));
byte myweekday = FILTERWEEK(getTimer(GET_DAY_WEEK));
```

`GET_MONTH`
`GET_YEAR`

`GET_SECOND` can be used as a base index, example to retrive the full date and time:

```
byte mytime[6];
for(i = 0; i < 6; i++) mytime[i] = getTimer(GET_SECOND + i);
```

Also with `DATETIME` the following macros can be used:

`GETCURTIME_M`	returns a `UINTVAR` of the current time, in minutes (seconds are ignored).
`GETCURTIME_S`	returns a `UNITVAR` of the current time, within the current hour, in seconds.

Example: `UINTVAR mycurtime = GETCURTIME_S;`

```
void setDatetime( int year,byte month,byte wday,
                  byte hour,byte minute,byte second)
```

Set date and time. Day can be incorrect and the function will try to automatically adjust the day. To combine day with weekday `setweekday(weekday,day)` can be used, however it is not necessary as just providing the day the function will automatically set the matching weekday.

Using time

Use the type `m_time` to declare a variable that must hold time. The global variable `gTime` hold the current time and can be used to make a copy of the current time. You should avoid to directly change this variable, though. Use setDatetime instead.

`m_time` is a union with two members:

`tc`	holds date and time as byte array
`t`	holds date and time with members: second, minute, hour, day and weekday, month, year (within century)

DAILY-WEEKLY SCHEDULER

The OS offer an optional scheduler for programmed operations. The following lists the relevant functions.

The scheduler is available only if enabled and only on devices that have time clock enabled and available as hardware component.

The scheduler handles scheduled events on daily basis over a weekly cycle. You can enter rules based on the time of the day, and for each day of the week. Each rule has a value that can be used either bitwise or as byte data. Per each time and day of the week up to four types of rules can be defined. All types work in parallel.
Rule types can be used to set weekly rules that apply to different operations. For example to control activations on different zones.

The scheduler works with the internal time, that you can use in conjunction with the timing functions `setDatetime` and `getTimer`.

The scheduler supports exception rules, that apply separately for each rule type. Exception rules take control on day-month basis. For example, you can enter an exception to the rule of type 1 that is valid from day 2 to day 25 of July, and set a specific value that apply while the exception rule is in force. For more information see `scheduler` event.

The scheduled data are stored in EEPROM, starting from the location defined by `PTREXCSCHED` which is where the pointer to the exception table is located. This two-bytes pointer points to the beginning of the day-month exceptions table.

Below is shown how the table of records is organized in memory. ET means End of Table bit, N means Null (zero).

pointer to the exception table		daily-weekly scheduler table	end of table							day-month exceptions table	end of table							
MSB	LSB		ET	N	N	N	N	N	N	N		ET	N	N	N	N	N	N	N
2 bytes		4 bytes per record	1 byte							4 bytes per record	1 byte							

The daily-weekly scheduler table is made of 4 bytes per record, with a single byte that marks the end of the table.

7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
N	N	minute						type				hour				N	Su	Sa	Fr	Th	We	Tu	Mo	value							
																	days														
byte 0								byte 1								byte 2								byte 3							

The day-month exceptions table is made of 4 bytes per record, with a single byte that marks the end of the table.

7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
N	N	N	start day					start month				end month				type			end day					value							
byte 0								byte 1								byte 2								byte 3							

SCHEDULER FUNCTIONS

```
byte getSchedRecord(unsigned char* index, byte type, unsigned char* rdata)
```

Retrieve a scheduler record of the given type.
Arguments:

`index`	pointer to progressive index of the record to read, this will be updated by the function if a matching record type is found.
`type`	type of record to read, it can be 0, 1, 2, 3. Only records of the required type are read.
`rdata`	pointer to an array of four bytes that will be filled with the record data

Return: 1 upon successful record read, 0 if the end of table is met and no more record are available.

Use example:

```
// retrieve all records of type 2:
byte ix = 0;
byte rec[4];
while(getSchedRecord(&ix, 2, &rec)) {
      // do something with the record
}
```

Remarks:
The updated index can be used with `deleteSched`. Index could be non-consecutive as different record types can be inserted in between. To dump the whole table use `readSched`.

```
byte getSchedException(unsigned char* index, byte type, unsigned char* rdata)
```

Retrieve an exception record of the given type from the scheduler.
Arguments:

`index`	pointer to progressive index of the record to read, this will be updated by the function if a matching record type is found.
`type`	type of record to read, it can be 0, 1, 2, 3. Only records of the required type are read.
`rdata`	pointer to an array of four bytes that will be filled with the record data

Return: 1 upon successful record read, 0 if the end of table is met and no more record are available.

Remarks: Scheduler exceptions cannot be deleted using deleteSched, nor new entries can be achieved using insertSched. You need to write the specific code to handle the exception table or use an external software to retrieve and send the table over the network.

```
void insertSched(byte value,byte hour,byte minute,byte stype,byte days)
```

Insert a new schedulation. The insertion is made automatically. Required values:

`value`	scheduled data value, it can be bitwise or an 'analog' value in range 0-255.
`hour, minute`	hour and minute at which the daily schedulation must happen over the week.
`stype`	schedulation type.
`days`	Bitwise, days when the schedulation apply. Bit 0 is Monday, bit 6 is Sunday.

Remarks: The function inserts a new schedulation in progressive order, only if no event exists. If an event having the same type, days and time already exists, the value is updated. If an event having the same time, value and type already exists then the days are updated. No new schedulation is inserted if another one with type, days, time and value already exists.

```
void deleteSched(byte index)
```

Removes the schedulation at `index`. If no event exists the function does nothing. To get a proper index the function `getSchedRecord` can be used.

```
void freezeScheduler(byte type, byte set)
```

Freezes the scheduler for the given type.
`type` type of programme to freeze/unfreeze, it should be in range between 0 to 3.
`set` 0 unfreeze, non-zero freeze.

The frozen schedulation type overrides the scheduled events and exceptions for that given type.

Comments.
The frozen schedulation becomes inactive but it is not stopped. If a scheduled event happens on a frozen schedulation, the event is ignored and not fired.
Past events happened while the schedulation was frozen are lost, but the next day (over the programmed days) may get a chance to be fired if the meantime the schedulation is unfrozen. This wouldn't happen if an exception event is met, though.

```
ResetSched;
```

Reset overrides and timetable events pointer, restaring the scheduler.

```
byte readSched(int location)
```

Read an item at the required `location` that can be any address from PTREXCSCHED to MEMSCHED. PTREXCSCHED is where the pointer to the table of exception is located, it is made of two bytes MSB first. The scheduler table begins from STARTSCHED.
Returns a data byte.

```
void writeSched(byte value, int location)
```

Write a schedulation byte into `location`, which can be any address from PTREXCSCHED to MEMSCHED. PTREXCSCHED is where the pointer to the table of exception is located, it is made of two bytes MSB first. The scheduler table begins from STARTSCHED.
Once written, the function also restart the scheduler.

NETWORKING FUNCTIONS

```
byte send(byte target, int data, int mask, byte type)
```

Send to target a packet.

target	address of the remote recipient of the message.
data, mask	application specific data.
type	type of message, see CTRLTYPES in the SFBP section below.

Return: The function returns either SENDRESULT_OK, SENDRESULT_BUSY or SENDRESULT_FAIL.

Messages are normally sent as connected packets.
If ctrlType is given as CTRLTYPE_USER_DATA the message is sent as a data message, otherwise is sent as control message.
The message is always sent as connected. If the recipient is BROADCASTADDR or ctrlType has CTRLTYPE_SYSTEM bit set, then the message is sent as datagram.
In case of datagrams no ACK is requied from the remote peer, so the function is faster even though the message could be lost.
If target is set to BROADCASTADDR then the packet is sent as datagram and no response is expected.

```
unsigned char SendCTRL(unsigned char dest,
                       unsigned short dat,
                       unsigned short mask,
                       unsigned char id,
                       unsigned char ctrlType)
```

Similar to send but allows to send a message ID as well. The message ID is used in RPCs and pubevents to transmit/receive the function's vector, but it can be used for other purposes when sending private messages.

dest	address of the remote recipient of the message.
dat	first data, this resolves into the first two bytes of the message.
mask	second data, this resolves into the following two bytes of the message.
id	message ID, it is sent as the fifth byte of the message.
ctrlType	this must be one CTRLTYPE.

If ctrlType is given as CTRLTYPE_USER_DATA the message is sent as a data message.
The message is always sent as connected. If the recipient is BROADCASTADDR or ctrlType has CTRLTYPE_SYSTEM bit set, then the message is sent as datagram.
With datagram messages no ACK is expected after they are sent over the network.

Return: The function returns either SENDRESULT_OK, SENDRESULT_BUSY or SENDRESULT_FAIL.

```
char RPC(byte target, byte vector, byte argcount, ...)
```

Execute a remote procedure call firing a pubevent.

`target` address of the remote device where the procedure must be executed.
`vector` vector of the procedure, this can be retrieved from the file vectors.h .
`argcount` number of arguments (see below)
`...` arguments: this can be none, one or up to four arguments as follow:
up to two arguments: each argument must be of type short;
three arguments: first one must be of type short, the following two must be of type byte;
four arguments: each argument must be of type byte;

Examples:

```
RPC(target, vector, 0);
RPC(target, vector, 1, short);
RPC(target, vector, 2, short, short);
RPC(target, vector, 3. short, byte, byte);
RPC(target, vector, 4, byte, byte, byte, byte);
```

The matching pubevent should accept arguments accordingly. See also pubevent.

Remarks: **Available only if `USR_USESETOUTVARG` is selected.** This function can be used in conjunction with `setsync` and `wait` (see Utiliy Macros for Networking).

```
SendData(sndaddr, opcode, data, len, packettype);
```

Send generic SFBP packet. This could be used to reply to a remote peer application specific command either in an `pubevent` or inside the `net` event to return one or more packets of data using the following syntax:

```
unsigned char mdata[DATABUFLEN];
// ... fill here mdata with some data ...
// send mdata back to the remote peer whose address should still be in the
// received packet: packet.srcAddr. Message is sent as datagram (not connected)
SendData(packet.srcAddr, OPCODE_DATA_PACKET, mdata, DATABUFLEN, TYPE_DATAGRAM);
```

UTILITY MACROS **FOR NETWORKING**

`setbusy` force busy status response for incoming requests.
`resetbusy` reset busy status.
`setsync` set synchronization before calling RPCs, RPC automatically clears this flag.
`resetsync` clears the synchronization.

`wait` wait for synchronized RPC completion.
This will call the underlying `_wait()` function, making the process to halt until a `CTRLTYPE_OK` message is not received from the remote peer or the busy flag is not cleared (see `resetbusy`), or a timeout (up to 8 second) happens. If timeout happens the function returns after having recorded an `errRPCFailed` error that can be read either using `Catch` or `getErr` functions (see Error Handling section).

`sendOK``(target)` send to `target` OK response. Use this if you want to free the sending device from waiting the completion of the called procedure. You don't need to use this macro as the OS automatically send OK back to the caller once the procedure exits.

SFBP - Networking

GLOBAL ADDRESSES

BROADCASTADDR	0	Broadcast send/receive
	1	Typical station address (i.e. computer or SCADA)
GW_ADDR	127	Gateway address
MAXADDR	128	First out of range address, max muber of addresses.

SEND RETURN VALUES

SNDBUSY	Network is busy, retry later.
SNDFAIL	Failed to send the message, either no response from peer or bad request.
SNDOK	Successfully sent the message over the network.

STANDARD CONTROL TYPES

Coded standard control types for control packet sent among units.
The control type is the last byte in a standard packet of DU length of 6 bytes. Based on the SFBP standard the meaning of Type is application related, and from that point of view the NSC O.S. is itself an application.
Therefore the type byte is shared between the OS and the user application, so that the CTRLTYPE_USER and the CTRLTYPE_USERD base types, by 84 user defined values, are available for user private purposes.

CTRLTYPE_USER_DATA	0	user defined or system packet with meaning of data packet (not ctrl packet)
CTRLTYPE_SETTARGET	1	set target where to send ctrl packets (see sendtarget as well)
CTRLTYPE_SETOUT	2	set output with received first two bytes as data and following two bytes as mask
CTRLTYPE_DUMPERR	5	send back the error stored in eeprom, if any
CTRLTYPE_GETIN	6	send back the last status of inputs
CTRLTYPE_GETOUT	7	send back the last status of outputs
CTRLTYPE_RPC_0PARAM	8	remote procedure call, zero parameters
CTRLTYPE_RPC_MAXPARAM	12	max count of arguments for RPC equal to CTRLTYPE_RPC_0PARAM+4
CTRLTYPE_SYSTEM	0x80	bit set used for system controls packet sent via SendCTRL with the highest bit set are considered as NON System but interpreted to be sent as datagrams.
CTRLTYPE_USER	43	user defined ctrl packets, connected - available range: CTRLTYPE_USER + 84
CTRLTYPE_USERD	141	user defined ctrl packets, datagram - available range: CTRLTYPE_USERD + 84

226 and up, system packets as following:

CTRLTYPE_IOSTATE	234	firmware version 1.19 and above will return I/O status if settarget is nonzero with this message instead of CTRLTYPE_INSTATE and CTRLTYPE_OUTSTATE
CTRLTYPE_RPCSYNC	235	set a flag to syncronize the following RPC
CTRLTYPE_FIRMWAREUPLOAD	236	
CTRLTYPE_POWERRESTORE	237	message which notify that power can be restored
CTRLTYPE_POWERSAVE	238	message to ask for power save
CTRLTYPE_RCLICK	239	remotable click
CTRLTYPE_SETSERIAL	240	set the serial code (only in non SMALL)
CTRLTYPE_GETSTORE	241	send back 5 byte of data stored in eeprom memory, beginning from the location specified by the first two bytes (LSB, MSB); the byte 5 returned hold the length of data (0 to 5)
CTRLTYPE_OK	242	sent in response to a sequenced operation such as the one required by CTRLTYPE_STORE
CTRLTYPE_STORE	243	require to store the following data packets in eeprom
CTRLTYPE_PRGSTATE	244	reprogram response, only if PFLASH is defined (program in flash)
CTRLTYPE_RSTADDR	245	reset address to null
CTRLTYPE_OUTVALUE	246	sent if settarget has been set to return the value of an output (i.e. dimmer)
CTRLTYPE_INSTATE	248	in response to getIn (or for setTarget) a packet with this type is returned
CTRLTYPE_OUTSTATE	249	on out change if settarget has been set, or in response to a getout, a packet with this type is returned
CTRLTYPE_GETSERIAL	250	send back the serial number and current address
CTRLTYPE_SETADDR	251	set address by serial number
CTRLTYPE_CLEARERR	252	clear error block memory
CTRLTYPE_REPROGRAM	253	set to program memory with next received data packets, data specify how many packets are expected
CTRLTYPE_REBOOT	254	reboot device, and stops the user program if first byte is set to 63 (and can only be resumed by clearing error followed by a reboot). If the first byte is set to 55, first stops and reboot, then (with a second message) it is fully stopped, that means no identification messages are sent over the network.

GRAPHIC LIBRARY

(When included for devices with display capabilities)

```
char* appendbuf(char* buf, const char* src, unsigned char start)
```

Append to the given buffer **buf**, at the **start** position, the string from the **src** constant buffer (in flash). If start position is 255 then from the given buffer the zero terminator is searched for and the src string is appended from there.
If buf is NOT initialized start MUST be given and set to zero.
Return: the modified buffer.

Comment: This function is useful if you want to manipulate a string that is located in flash. Use `appendbuf` either to copy or append into the given buffer one or more strings.

```
unsigned char flashRead(unsigned int i)
```

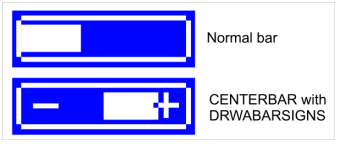
Read from flash at the given location i.
Return: a byte read from flash.

```
void drawbar(unsigned char size, unsigned char div, unsigned char row, unsigned char col, PRNTVALUE value)
```

Draw a bar gauge
This function is available only if `DRAWBARS` is selected or defined. It is possible to specify `HORIZONTALBARS` or `VERTICALBARS` to enable only one of the two modes, making the code slighly smaller.

Arguments:

`size`	Width or height of the bar, must be 1 to 20 characters or 1 to 6 rows. It can be combined with `CENTERBAR` or `VERTBAR`.	
`row, col`	Position of the bar, in characters.	
`div`	`DIVBYTE`	255 if the value is between 0 to 255;
	`DIVPERC`	100 if value is between 0 to 99 (0-100%)
`value`	Value of the bar, it could be in range between 0 to 255, or 0 to 100%, or in pixels (compared to the width of the bar).	

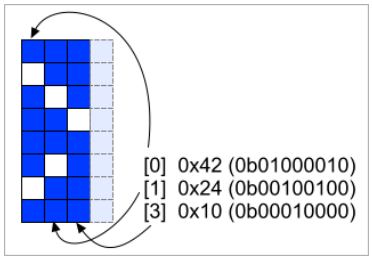


`PRNTVALUE` is a short (16bit) or long (32 bit) if `PRINTLONG` option is set.
Argument `size` can be combined with `CENTERBAR` (ex.: `width | CENTERBAR`) to create a centered bar. In such a case `value` is expected to be a signed integer, example from -50 to +49.
Alternatively size can be combined with `VERTBAR` to draw a vertical bar. Vertical bars cannot be with centered zero, if you combine both `CENTERBAR` and `VERTBAR` the result is unpredictable.

If `DRAWBARSIGNS` option is selected as well, bars with `CENTERBAR` are also drawn with + - signs.
If `value` is in range between 0 and 255 then `DIVBYTE` should be given to `div`; if it is in range of 0 - 100 then `DIVPERC` should be given to `div`; if it is in pixels the range should be the same value given for `size`.

```
void drawPixels(unsigned char* data, unsigned char arraysize)
```

Available only if DRAWBARS and VERTICALBARS are defined. Draw at the current cursor position a memory bitmap. The bitmap cannot span beyond a single row, but can span multiple characters. To draw over multiple rows, send an array of bits per each row.
The bitmap must be provided as a series of vertical lines of pixels. Each byte represent a vertical line of 8 pixels (see figure).
The function does not change the cursor position.




```
void movecur(unsigned char row, unsigned char col)
```

Move the cursor to the given row and column, in characters.

```
unsigned char GLgotoX(unsigned char p)
unsigned char GLgotoY(unsigned char p)
```

Respectively move the current cursor position to p along X and Y axis, in pixels.
Return: 1 success, 0 the required position is outside the allowed space.

```
void GLbitmapOut(tagGLbmpOutInfo* bi)
```

Draw a bitmap
Arguments:

bi	BitmapInfo structure:
adrBmp	address in flash where the bitmap bits are stored.
x, y	coordinates where the picture must be placed on the display.
width, height	The size of the bitmap. Only width by height bits will be read from the bitmap bits file and blitted out to the display. If width is set to zero the bitmap width and height are read from the file, and the adrBmp should point not just at the beginning of the bitmap bits, rather at the beginning of the bitmap file.

```
void _putchar(unsigned char ch)
```

Print a character to the LCD at the current location.

```
void textOut(char* t, const char* tf, ...)
```

Draw a text at the current X,Y position
Arguments

`t`	pointer to a buffer which hold the text to draw, this member can be a `NULL` pointer if the `tf` member is provided. Strings pointed by `t` must be of type ASCIIZ (null-terminated).
`tf`	If `t` is `NULL`, `tf` must be a const pointer to a flash string containing the text to draw. If this member is zero, then the `t` member must point to a valid buffer. The string pointed by `tf` must be of type ASCIIZ.
...	Optional variable number of arguments (see notes).

Remarks:
The text is drawn at the position set by the `GLgotoX/Y` functions, if a `LF` character is met, the line is reset to the current X position as it was at when calling this function (no reset to zero). This leave the text aligned to the left, where 'left' is the last X position.

Notes:
If the string to print expects arguments you should provide them after `tf`, in the same order as required by the string.
Additional arguments are required when using:

type of element embedded into the string	number of arguments	argument type(s)
embedded indexed strings	1	`(unsigned) char`
selectable items, to indicate the currently selected item	1	`(unsigned) char`
numeric placeholders	1	`PRNTVALUE`

Important: you should provide the correct variable type, internally the function extract the argument for the type as listed above.
If your data is signed or unsigned cast the value into the required format.
Example, print an unsigned short data (it is assumed the string has an absolute numeric placeholder):

```
unsigned short data;
data = 44500; // assign a value to data
textOut(NULL, pMyString, (PRNTVALUE)data);
```

CAUTION: textOut requires `USR_USESETOUTVARG` option!

WARNING: TO SAVE MEMORY NO CHECK IS DONE HERE HOWEVER YOU SHOULD NEVER CALL THIS FUNCTION WITH NUMERIC OR STRING PLACEHOLDERS OR ESCAPE FORWARD EMBEDDED CONTROL CHARACTERS IN THE STRING TO PRINT BY GIVING AN ASCIIZ POINTER SINCE THIS OPTION IS AVAILABLE ONLY WITHIN THE SYSTEM CODE.

```
PRNTVALUE convert10Bit2Decimal(unsigned short value, unsigned char options)
```

This function is available only if ANALOG10BITS is selected, thought it is available even if no analog inputs are enabled.

Converts the given value expressed as 10 or 12 bits integer into a proportional decimal value. This is useful to convert data read from ADC either at 10 or 12 bits.

This function compile differently whether HIPRECISION_HIRES_CONVERSION is defined or not. If this option is defined the code takes about 80 more bytes once compiled, but improves the conversion error.

Conversion errors:

option	*other options*	*Range and conversion errors*
Default	-	Value range: 0÷1023. Max conversion error: 2% Max error over the full range: 0.01%
HIPRECISION_HIRES_CONVERSION	Default	Value range: 0÷1023 Max conversion error: 0.2% Max error over the full range: 0.01%
	OPTCONVERT_12BITS	Value range: 0÷4092 Max conversion error: 0.33% Max error over the full range: 0.03%
	OPTCONVERT_12BITS + OPTCONVERT_499	Value range: 0÷4092 Max conversion error: 1.32% Max error over the full range: 0.04%

options selects the conversion reference:

OPTCONVERT_UNSIGNED	0	Converts and returns the given value as unsigned integer (default).
OPTCONVERT_SIGNED	1	Converts and returns the given value as signed integer where zero is at the middle: 512 for 0÷1023 range (10 bits); 2046 for 0÷4092 range (12 bits); The returned value is thus in range -5000 ÷ 0 ÷ +4999 .
OPTCONVERT_12BITS	2	Evaluates the given value as 12 bit integer (from oversampling and decimation). The accepted value is thus in range between 0 and 4092. Without this option the accepted range is up to 1023.
OPTCONVERT_499	4	Returns the given value converted from 0÷4092 to 0÷5000. It requires OPTCONVERT_12BITS . If OPTCONVERT_SIGNED is also defined the result is a negative range: -5000 ÷ -1 .

Return:

The function returns a four digits and half decimal signed integer value comprised between 0 and 9999, or +4999 and 0 and -5000 if OPTCONVERT_SIGNED is given as option.
The returned value can be passed to ***textOut*** or ***putnum*** with mode = MODENUMCENT or MODENUMDEC or MODENUMMILL (millesimal) .

```
void putnum(PRNTVALUE value, unsigned char mode)
```

Put a number on screen at the current cursor location.
Parameters:

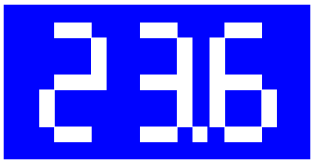
value	the numeric value to display
mode	display mode:

MODENUMNORMAL	0	default mode, display number as is
MODENUMABSZERO	0x80	*Absolute* representation, do not show sign (will void the place required for sign as well, so the number is tightly aligned to the left). If combined with a digit count, unfilled digits on the left are filled with leading zeroes. Not compatible with MODENUMHALFBYTE.
MODENUMABSSPACE	0xA0	*Absolute-space* This option is a combination of 0x20 and 0x80, making it incompatible with MODENUMHALFBYTE. Same as absolute described above, however unfilled digits on the left are filled with spaces and not with zeroes.
MODENUMHALFBYTE	0x20	*Half byte*. Converts number from 0 to 127 as negative number from -500 to 0; and from 128 to 255 as positive number from 1 to 499. The number can be displayed with one, two and three decimal digits. This mode has meaning only for values in a byte range. NOT compatible with 0x80 *Absolute* option (see 0xA0 above).
MODENUMBYTE	0x40	*Byte to decimal (255/10)* range display mode, the number is converted from its range of 0-255 to the related range of 0 - 999, the number can be displayed with one, two and three decimal digits. Useful to display voltage values read from the analog inputs.
MODENUMDEC	0x08	show number with one decimal (as if the value is divided by 10).
MODENUMCENT	0x10	show number with two decimals (as if the value is divided by 100).
MODENUMMILL	0x18	show number with three decimals (as if the value is divided by 1000). Note this is the combination of MODENUMDEC and MODENUMCENT mode flags.
Digits Last 3 LSbits:	0 to 7	*Digits* (NUMFILTERDIGITS) fixed digits count, this defines a space to show the number keeping it aligned to the right in relation to the number of digits defined by this member, void places to the left are filled with blanks (deleting the background). The count does include the fixed place for the sign, if allowed. Therefore if absolute mode is set, the count must be one less then what it would be if no absolute mode were set. If this is zero then no padding is made to the left of the number, and the number is printed aligned to the left.

Remarks:
Various modes may be combined, but the MODENUMABSZERO, MODENUMABSSPACE and MODENUMHALFBYTE options that are mutually exclusive. The MODENUMHALFBYTE is not compatible with numbers that not fall within the range of 0 to 255.
Absolute (unsigned) mode, removes the default blank on the left side of the number that is otherwise inserted for positive numbers (if SHOWPLUSSIGN is defined a + is shown in place of blank).
Combining *Digits* and *Absolute* will display the number with leading zeroes when the number is smaller for the given number of digits.

Combining `MODENUMDEC, MODENUMCENT, MODENUMMILL` makes the number to be shown with decimals. Digits and decimals can be combined as well. Decimals can be combined with all other modes.

If **`SEVENSEGMENTS`** option is selected then numbers with attribute `_BOLD_` are printed as seven segment digits (no plus sign is printed, though). The size of each digit takes twice the size (both in height and width) of regular characters, decimal point is embedded in digits so it doesn't take any space.



Example:

```
// Print value from DS1820 temperature sensor.
// Multiply by ten to get temperature with
// one decimal, divide by two because DS1820 returns value in half degrees.
// Display mode with one decimal digit, and force to take space
// for almost 6 digits.
putnum((PRNTVALUE) getIn(DGTDATA0)*10/2, MODENUMDEC | 6);
```

```
clearline(startX, endX, line)
```

This macro calls the internal function specific of LCDG:
```
void GLclearline(unsigned char startX, unsigned char endX, unsigned char line)
```

Clear a line from `startX` to `endX`, in pixels for the given `line`. If this function is called after the $invert or $underline options are set, the line is cleared inverted and/or underlined.
On 64x128 displays `startX` and `endX` must be between 0 and 127, while line must be from 0 to 7.
On exit the function moves the cursor at zero.

```
void cls(void)
```

This calls the internal function specific of LCDG:
```
      void GLclear(void)
```

Alias: `LCD_CLEAR`
Cleare the whole screen. Also it resets all format settings.

```
void showicon(int iconID, byte X, byte Y)
```

Draw the icon at the `iconID` location in flash, at the `X` and `Y` position on screen. The position must be in pixels, from 0 to 63 for Y and from 0 to 127 for X.

LCDG SPECIFIC

```
void drawicon(unsigned int id)
```

Draw an icon at the current cursor position. **`id`** must point to the address of a valid `tagGLbmpOutInfo` in flash.

```
char* cpybuf(char* buf,const char* src)
```

Copy into the given buffer `buf` the data from an ASCIIZ string pointer `src` from flash.

Global modifiers

SetUnderline	Set to print text underlined. All subsequent calls to textOut, putchar and putnum are affected. Use SetNormal to remove this setting.
SetDotsUnderline	As above but underlines are dotted. If NEGATIVEUNDERLINE option is defined this will print overlines in place of underlines.
SetBold	Set to print bold (enlarged) text. All subsequent calls to textOut, putchar and putnum are affected. If SEVENSEGMENTS is defined putnum will display large, seven segment display style numbers.
SetInvert	Set to print inverted colors text. All calls to textOut, putchar, putnum are affected.
SetNormal	Reset all settings and print normal text. textOut, putchar and putnum are affected.

STRING COMMANDS AND STRUCTURES

Commands to display the selected item in a itemized print

escape sequences (to be inserted into strings):

_RESET_ITEMSCOUNT_ requires one argument given to the function textOut to provide the currently selected item, usually put at the beginning of the itemized string.

ITEMINVERT
ITEMUNDERLINE
ITEMRIGHTARROW

binary representation:
B_RESET_ITEMSCOUNT_
B_ITEMINVERT_
B_ITEMUNDERLINE_
B_ITEMRIGHTARROW_

Indexed strings points to a string from gTextCollection array and the provided argument (to textOut) offsets to the next strings in the array. The value passed to the macro must be one based and as hex value as string, i.e. "\x01" which points to the first string in the array.
(escape sequences:)
_INDXSTR(id)
INCSTR

Move cursor
Move cursor, is made by \x1b followed by one digit for the row and one or more digits for the column followed by semicolon, i.e.: \1b018; would move at row=0 and col=18
_MOVECUR(row,col)

Clear a line of text
CLEARLINE

NUMBER PLACEHOLDERS

`digits` is the number of digits to show, or an encoded string that provides the number of digits and the number of fixed digits (plus sign).
This string must be entered as a decimal value for simple number of digits (i.e., for `_NUM_CENTABS`), or as a hexadecimal value where the identifier requires a composed string: example `_NUM_ABS_ZEROPAD(\x44)` indicates four fixed digits and two decimals, encoded.

escape sequences (to be inserted into strings):

```
_NUM_NORMAL(digits)
_NUM_DECIMAL(digits)
_NUM_CENT(digits)
_NUM_HALFBYTE(digits)
_NUM_BCD1K(digits)
_NUM_BCD100(digits)
_NUM_BCD10(digits)
_NUM_ABS_ZEROPAD(digits)
_NUM_DECABS(digits)
_NUM_CENTABS(digits)
_NUM_ABS_SPACEPAD(digits)
_NUM_HEX(digits)
_NUM_MILLESIMAL(digits)
```

Hex and Millesimal are alternatively interpreted depending by whether `SHOWHEXNUMFORMAT` is defined or not.

Structures

```
typedef struct
{
 unsigned int adrBmp;
 unsigned char x;
 unsigned char y;
 unsigned char width;
 unsigned char height;
} tagGLbmpOutInfo;
```

`x, y, height` and `width` in pixels, `adrBmp` is the address in flash.
Bitmaps must be stored with the XBM format.

ERROR HANDLING

```
void Try(void)
unsigned char Catch(void)
```

Used to catch possible errors that certain functions may raise by calling raiseException (see below). `Catch` returns the raised error, or zero if no error was raised, and resets the `Try` block. The error buffer is not cleared. See also `getErr`.

```
unsigned char getErr(void)
```

Retrieve the last error code, zero if no error. Calling the function also clears the error buffer. See also `Catch`.

`Catch` or `getErr` are useful with RPCs to check for any possible error.

```
void raiseException(unsigned int errorCode)
```

Custor exception raised. This function raises an error and `errorCode` is stored into the error buffer. If a `Try` function was called before calling `raiseException` then if the `errorCode` is greater than `errUNRECOVERABLE` the function simply stores the `errorCode` into the error buffer.
Otherwise in both cases where `Try` was not called or `errorCode` is smaller or equal to `errUNRECOVERABLE` the function also halts and restart the processor which will enter into an alarm status that must be cleared (using a `CTRLTYPE_CLEARERR` message) before restarting again normally.

You can use this function to record an error or a severe error that should halt the operations and bring the user's attention to the device by checking its error calling `CTRLTYPE_DUMPERR` .
Once the device entered into an unrecoverable error it won't restart normally again until a `CTRLTYPE_CLEARERR` message is not received.

Error list

`errStackOverFlowed`	7	
`errRcvProgramError`	8	
`errFLASHFAIL`	9	
`errINVALIDPROGRAM`	11	
`errHwStackOverflowed`	12	
`errFLASHFAILURE`	13	severe flash failure (read back written data doesn't match current data buffer used to write)
`errUNRECOVERABLE`	19	max error number below of it any error is considered as unrecoverable
`errRPCFailed`	21	
`errREADDRESS`	240	not an error, set when a readdressing is issued, this error is reset if a new program is then uploaded
`errSTOPPED`	241	device is full stopped, so no signal is sent when I/O changes occurs, anyway this value is never stored in the error registry in eeprom, so that it is cleared as soon as a reboot is called.
`errVirgin1`	254	same as virgin, but serial code has been set. It means never progrogrammed.
`errVIRGIN`	255	this mean no error was written to eeprom and is interpreted as VIRGIN device!

API - EVENTS

All listed events are available only if selected in AppWizard.

EVENT_CLICK

```
event click(unsigned int Clicked)
```

Fired when one or more logic inputs go up and then down within 2 seconds.
`Clicked` 16 bit array representing the inputs that have experienced the click event.

You can use `TESTIO` to check which input has been clicked, example: `TESTIO(Clicked, IN1)`.
This event is fired even when a 4 - 5 keys keyboard is connected (see Keyboard macros)

EVENT_LONGCLICK

```
event longclick(unsigned int Clicked)
```

Fired when one or more logic inputs go up and stay up for longer than 3 seconds, and then go down.
`Clicked` 16 bit array representing the inputs that have experienced the longclick event.

You can use `TESTIO` to check which input has been clicked, example: `TESTIO(Clicked, IN4)`

EVENT_INPUT

```
event input(unsigned int IN, unsigned int HIN, unsigned int LIN)
```

Fired when one or more logic inputs have changed.
`IN` 16 bit array representing the status of each input.
`HIN` 16 bit array representing the inputs that have gone up.
`LIN` 16 bit array representing the inputs that have gone down.

You can use `TESTIO` to check which input has been clicked, examples:

```
if(TESTIO(IN, IN4)) /* check whether input 4 is high or low */;
if(TESTIO(HIN, IN2)) /* check if input 2 has changed up */;
if(TESTIO(LIN, IN0)) /* check if input 0 has changed down */;
```

Caution: this event is always fired when an input changes, even if the event is then deemed to be click or longclick.

EVENT_NET

```
event net(int data, int mask, byte id, byte type, byte sender) (with EXTENDEDNET)
event net(int data, int mask, byte type, byte sender)
```

Network event. The first version is available only if `EXTENDEDNET` is selected.
Fired when the device receives a valid packet targeted to its address, or a broadcast message.

`data` first couple of bytes of data: first byte LSB, second byte MSB.
`mask` second couple of bytes of data: first byte LSB, second byte MSB.
`id` (only with `EXTENDEDNET`) message ID (typically sent with RPCs).
`type` message `CTRLTYPE`.
`sender` address of the remote peer that sent the message.

EVENT_OUTCHANGE

```
event outchange(int changed)
```

Fired when one or more logic outputs change, either because of the user program or because a remote peer commanded to turn on or off one or more outputs.

`changed` 16 bit representation of the outputs. Changes on virutal outputs shall fire this event as if they were actual physical outputs.

You can use `TESTIO` to check which output has changed, example: `TESTIO(changed, OUT3)`

EVENT_BUTTON

```
event button(byte btn)
```

This event is available only if LCDH (non graphic LCD device or equivaled device with 12+ keys keyboard, not to be confused with LCDGH which refers to the graphic LCD) is selected.
Fired when a button is pressed by the user.

`btn` index of the pressed button

Rem: This event is not actually fully implemented.

EVENT_TIMER0 TO EVENT_TIMER7

```
event timern(void)      where n is 0 to 7
```

Fired when the related timer expires.
Remark: Calling `setTimer(0, TIMER`*n*`)` (where *n* is 0 to 7) cancel the timer if it is not yet expired.

EVENT_ANALOG0

Available only if **ANALOGEVENT** is selected.
BEWARE analog events may be called at a steadfast frequency!

```
event analog0(int channels)
//event analog1(int value) deprecated, this event is no longer available.
```

Fired when the read value from the related analog input goes beyond the set threshold, be it above the `higherLevel` or below the `lowerLevel`. See also `setAnalogTrigger` .

Alternatively this event is fired when one of the analog input changes if **ANALOGLEVEL** is selected in place of **USEANALOGTRIGGERS**.

The argument `channels` is a bitwise that provides the channels of the analog inputs that caused the event.

Example:

```
event analog0(int channels) {
      unsigned int data;

      if(channels & AW0) { /* channel 0 changed */ }
      if(channels & AW5) {
            /* channel 5 changed, get its data */
            data = getIn(ANDATA + AW5);
      }
}
```

Remark: The argument is cleared once the event return.

EVENT_INTERRUPT0, EVENT_INTERRUPT1,EVENT_INTERRUPT2

Available only if `INTERRUPTASEVENTS` is selected.

```
event Interruptn(unsigned int IntInputs)  (where n is 0 to 2)
```

Fired when the corresponding interrupt is triggered.

`IntInputs` provides the status of the inputs subject of interrupt, when the interrupt had happened.

EVENT_ZEROENCODER

```
event zeroEncoder(void)
```

Available only if `HASENCODER` is selected, and if the device support the encoder.
This event is fired when its trigger is set and the Z channel of the connected encoder goes down. This means the encoder must provide a normally high output, falling when the zero position is reached. To set the zeroencoder trigger call *encoder*`(RESETONZERO);` function.

Remark: When `HASENCODER` is defined both interrupt 0 and 1 are used for reading the signals B and Z (respectively) coming from the encoder. The *interrupt0* is set to be fired at each variation of the input, while *interrupt1* is set to be fired when the input goes low.

See also: `encoder` function.

EVENT_SCHEDULER

```
event scheduler(  byte type_exc,
                  byte type_clr,
                  byte value0,
                  byte value1,
                  byte value2,
                  byte value3  )
```

Available only if `SCHEDULER` is selected.

This event is fired when a scheduled event happens. Because multiple events could happen for different rule types, a value is provided for each type.

Arguments provided:

`type_exc` — Rule type, and exception type. Up to four types may happen. Each bit represent a type, the upper 4 bits represent the exception types while the lower 4 bits represent the scheduled rule types. If the bit is set it means the related rule or exception has occurred.
Keep in mind that more than one type could happen at the same time, with each own value (see below).

bit 7	6	5	4	3	2	1	0
excep. type 3	excep. type 2	excep. type 1	excep. type 0	rule evt. type 3	rule evt. type 2	rule evt. type 1	rule evt. type 0

`type_clr` — Exception cleared event. Each bit represent an exception type that has expired (the exception is no longer in force). Only the upper 4 bits are meaningful. If the bit is set it means that the related exception type has been cleared. The related provided value is the one that was there before the exception occurred.

bit 7	6	5	4	3	2	1	0
excep. type 3 ended	excep. type 2 ended	excep. type 1 ended	excep. type 0 ended				

`value0` to `value3` — Rule or exception value. `value0` is for type 0, `value1` is for type 1, and so on. The provided value is meaningless if there is no bit set for the related rule, exception or cleared type.

Remarks: For a given type, if its bit is set on more than one section then exceptions override rules, and rules override cleared exceptions. So for example if `type_exc = 0b10001000` then `value3` is meaningful but as exception, and the exception status prevails.
If `type_exc = 0b00001000` and `type_clr = 10000000` then `value3` contains the value of rule type 3, since the rule prevails on the cleared exception. However `type_clr` provides information that the exception type 3 has ended as well.

EVENT_POWERDOWN

```
event powerdown(void)
```

Fired when the device detects a power down condition or a powersave message is received.
If the device is already in powersave mode the event is not fired.
Once this event is fired the device is already in power save mode, which means the relay outputs are released to save power, however the output status is preserved in memory. If any change occurs while in power save the relay outputs won't change but the changes are nevertheless memorized so they can be restored when the power comes back.
While in power save the output change notifications are ignored.

EVENT_POWERUP

```
event powerup(void)
```

Fired when the device detects a power up condition. This means the power is went back to normal level continuatively for almost 15 seconds, or the device receives a power restore message.
This event is not fired if the device is not in power save mode, and when this event is fired the device has already exited from power save mode. When this event happens the outputs are restored to the status they should have had, in other words the outputs are restored taking in account of the changes occurred while in power save mode.

Remark: Powerdown and powerup events are available on certain devices only and when powered both via mains and battery. The event is fired when the voltage goes below (or above) 18V, but remains above 6V via batteries.

EVENT_KEYDETECTED

```
event KeyDetected(byte k0,byte k1,byte k2,byte k3,byte k4,byte k5)
```

Available only if ONEWIRE and IBUTTONPIN are selected.
Fired when a iButton key is placed on its receptacle and recognized by the device. The event provides the key as six bytes argument from k0 to k5.
The provided arguments can be used to match against one ore more stored keys.

EVENT_TCHANGE

```
event TChange(int temperature,byte item)
```

Available only if `ONEWIRE` is selected.
Fired when a temperature change of almost 0.5°C happens.

`temperature` temperature read by the sensor.
`item` number of sensor, on single sensors this member is always zero.

This event is available only when digital temperature sensors DS1820 are used (over OneWire).
The provided temperature value must be converted. For DS1820 sensors the conversion follow this formula: `int T = temperature * 10 / 2;` This way the converted value is multiplied by ten, so for example 21.5°C is converted into 215.
The value of 170 (0xaa) means the sensor is faulty or not responding or not connected.
Because the sampling frequency of the sensor happens at a pace of 20 second, this event won't be fired at a shorter interval.

PUBEVENTs

Public events (**pubevent**) are events that can be fired through RPCs or just by calling them into the code, as for normal events or normal functions. In fact you can call even the events into your code, as they are just functions that return void.

Pubevents are private, application specific functions that you can implement into the PUBEVENTS SECTION into the *devicename*.evt.c.h file (§ PUBEVENTS IMPLEMENTATION).

Example

Suppose we have two devices, one named DisplayU and the other named CtrlU.
From CtrlU upon the event input the public event inside DisplayU is called, sending the status of the inputs. At DisplayU the public event provides a notification on screen printing the status of the inputs from the remote unit CtrlU.

Inside DisplayU we've implemented the public event ActionTwo, which accepts two arguments.

```
////////////////////////////////////////////////////////////////
// PUBEVENTS IMPLEMENTATION
////////////////////////////////////////////////////////////////

pubevent ActionTwo(int arg1, int arg2, byte caller)
{
      // print on screen the string actiontwo_received
      // which in turn prints also two numbers
      textOut(0, actiontwo_received, arg1, arg2);
}
```

Inside CtrlU into event input we've implemented the remote call to DeviceU, giving the status of the input that changed high, and low.

```
////////////////////////////////////////////////////////////////
// EVENTS IMPLEMENTATION
////////////////////////////////////////////////////////////////

#ifdef EVENT_INPUT
event input(unsigned int IN, unsigned int HIN, unsigned int LIN)
{
      // call ActionTwo on DisplayU
      RPC(DisplayU_ActionTwo_VECTOR, (int) HIN, (int) LIN);
}
#endif
```

CUSTOMIZE YOUR DEVICE

Into the NSC-SDK folder you may find the folder Devs, inside it you can find a file named *usertemplate.c.h* .
Use this file as a boilerplate to create your own device file, simply changing the code where specified. In short this file tells how GPIO are mapped, including for special functions such as One-Wire or Encoder.
In addition you have to change the GPIO input/output, direction and pullup initialization, and the HAL PortIN and PortOUT functions that translate the GPIO into the Status Input and Status Output 16 bit values, so that the physical INx or OUTx match with the logical INx or OUTx into the program.

Give to your file with a proper name that must end by ***.c.h*** and save it into the UserDevs folder.

Once you prepared your device file you need to add the device to the list of available devices.
Start ***NSC App Wizard***, click button New Model.

Into the form that pops up enter a unique device ID for your new device (start from 100, up to 255 you have room for 155 devices). Enter a device name, a name or code of your device's hardware, and an optional description.

Into the field device file enter the filename and full path of the file you just saved, example:
C:\CHG\NSC-SDK\UserDevs\mynewdev.c.h

Then select the capabilities of your device, in other words what it supports.

Select the option you want to make available.

Finally enter an optional pipe (|) separated list of fixed defines required by your program for this device. Suppose your device need a couple of fixed defines '`MYDEF1`' and '`MYDEF2`', enter here (without spaces): MYDEF1|MYDEF2

If the device has analog inputs, click button Analog Channels to enter the number of available analog channels.

To complete the operation click Create/Save.

Now your new device is available on the list of devices of ***AppWizard***.
If you can't find the device in the list it could be the list was locked: Close and restart the application to allow it to release the list and reload it.