Programming Manual

Lua Standard Application API CLxNX

Copyrights

The contents of this document are proprietary information of SATO Corporation and/or its subsidiaries in Japan, the U.S and other countries. No part of this document may be reproduced, copied, translated or incorporated in any other material in any form or by any means, whether manual, graphic, electronic, mechanical or otherwise, without the prior written consent of SATO Corporation.

Limitation of Liability

SATO Corporation and/or its subsidiaries in Japan, the U.S and other countries make no representations or warranties of any kind regarding this material, including, but not limited to, implied warranties of merchantability and fitness for a particular purpose. SATO Corporation shall not be held responsible for errors contained herein or any omissions from this material or for any damages, whether direct, indirect, incidental or consequential, in connection with the furnishing, distribution, performance or use of this material.

SATO Corporation reserves the right to make changes and/or improvements in this product and document without notice at any time.

Trademarks

SATO is a registered trademark of SATO Corporation and/or its subsidiaries in Japan, the U.S and other countries.

Version: STL00249PB1

© Copyright 2018 SATO Corporation.

All rights reserved.

Table of Contents

1

Introduction

1.1 Functions

Lua is extended with the firmware API (STB00011) that controls fundamental properties of the printer like file system, communication, fonts, barcodes and the printer mechanics. On top of this is the Standard Application (SA), forming the abstraction, within which the user is performing his tasks.

This document describes the SA API library that, together with f/w API, can be used to extend SA with Functions. Functions are executable Lua code that is used to customize the application. Interaction with other resources like Formats, Tables and Images can be comprehensive and as such be considered as separate applications that reside in the printer.

A **placeholder** is a position in the SA environment that can be populated with any Function.
If populated then it provides a **callback** that can be used to change the default behavior.

1.2 Pipe Process

An example is the data **pipe process.** Each format field has a data source with predefined behavior except the source script where any function can be used. The value of any source is processed in 4 more steps with placeholders called prescript, round, formatter and postscript. It is rare that all steps are populated. The formatter divides the pipe process in two parts where Value becomes a formatted string called Data.

Example: If source is time then Value is in seconds and can be used for calculation but after the formatter, when seconds has become a localized form of date, then it is more suitable for printing.

1.3 Named Functions Callbacks

SA is also provided with callbacks that are located in positions where it possible to customize SA. Such callbacks should not interact with rendering and format data process since they are not executed during design time in AEP Works. One example is the Start callback that is called after power up. (See below)

2

SA reserved names

2.1 Namespace

To avoid interference with SA, customer added code is executed in a separate namespace (i.e. Lua module) called “safe environment” (se). From se all global (variables and library functions) in SA can be accessed directly. However if a global variable name is the same as a variable in se then then that variable will be used. Similar to when a function defines a local variable with the same name as a global. There are situations when these borders need to be crossed. To access a variable in se use “se.” as prefix. Global namespace prefix is “_G.” must be used to define names in global space.

Normally all global variables and functions in f/w and SA use “camelCase” notation. This means that the first letter is lowercase. (Exceptions to this rule are reserved SA API variables listed below.) **To avoid interference it is therefore recommended that user defined names starts with an uppercase letter**. Furthermore it is a good practice to use local variables to avoid pollution and increase speed.

2.2 SA API variables

SA variables are reserved global names for interface between SA and se environment.
They are used for direct access to actual values from selected Table, selected Format and actual Field that is executing.

Variable	Type	Description
Table	table	Selected or associated table
Row	table	Selected row in Table with column names as attributes
Rows	table	Latest selected row in table Rows.<tablename>
Format	table	Selected or associated format with field names as attributes
Formats	table	Accessed format Formats.<formatname>
Field	table	Format field that is interpreted
Value	Any	Field.value (pipe process)
Data	string	Field.data (pipe process)
Pfs	table	Power fail storage
Error	string	Error in pipe process
Canvas	object	Current print image
resolver	function	Present when executed from AEP Works

These variables can be used to access data in scripts.
(For available attributes see STL00255 SA XML CLxNX.doc)

When a table is accessed a dot notation can be used if the key is an alphanumeric string. Otherwise brackets must be used. See below

Syntax: Row.<ColumnName> or with brackets Row["<Column Name>"]
If the attribute contains spaces or other delimiting characters then brackets must be used.

Examples:
Table.name refers to the name of the associated table.
Row is the selected row in Table
Row.Price refers to the cell pointed to by selected row and Price column in Table.
Format.name refers to the name of the active format.
Format.Shop.value refers to the actual value of the format field with the name Shop.
Field.dateformat refers to the attribute dateformat in the actual format field.
Rows[“Shoe table”] is the latest selected row in table with name ”Shoe table”.
Formats[“Price Demo”] is the format with the name “Price Demo”.

2.3 SA reserved Function names (callbacks)

Some Function names are reserved for callbacks that are used for customization. These names are defined in the (se.) global space.

2.3.1 Start

Start is called after power up and can contain various functions and a priori settings.
Functions (and callbacks) does not need to have a separate file, they can all be defined in Start.
(See cbTranslate, cbWelcome etc. below)

2.3.2 cbMenuFormat

cbMenuFormat is called just as a format-based application has been selected.

cbMenuFormat (tFormat)

Variable	Usage
tFormat	This is the Lua table that represents the Format.
	This function does not return anything. Normally used for manipulating the format.

2.3.3 cbMenuTable

cbMenuTable is called just as a table-based application has been selected.

cbMenuTable (tTable)

Variable	Usage
tTable	This is the Lua table that represents the Table.
	This function does not return anything. Normally used for manipulating the table.

2.3.4 cbBatchDone

cbBatchDone is called when a print batch is finished

cbBatchDone (sMode,tFormat,nAdd,nDone,nAbort,tChain)

Variable	Usage
sMode	sMode contains “ONLINE”,”NORMAL” or “COPIES” or “CHAIN”
tFormat	This is the Lua table that represents the Format.
nAdd	Number of added print jobs.
nDone	Number of successful print jobs.
nAbort	Number of aborted print jobs.
tChain	History when printing Group of Labels. (CHAIN)
	This function does not return anything. This function can be used to do print logging.

Table with Group of Labels history:

```
if tChain then
  for i,v in ipairs(tChain) do
    dprint(v.name, tChain[v.name].add, tChain[v.name].done, tChain[v.name].abort)
  end
end
```

2.3.5 cbTableDisplay

cbTableDisplay is called before the database rows are displayed and gives an opportunity to design how the data is displayed.

cbTableDisplay (name, column, rows, searchKey)

Variable	Usage
name	Table name. (So called friendly name, not the file name)
column	Table display column name.
rows	Indexed Lua table with one Lua table for each row. Modify this table to modify what is displayed.
searchKey	The key used to find the rows. This parameter is nil if used in older versions. It is nil if searching in predictive mode. As table searching returns multiple rows, this parameter, when given, can be used to remove rows that are not matching.

Ex: Concatenates column 3 and column 1

```
function(name, column, rows)
  if name=="Shoe table" then
    for i,col in ipairs(rows) do col[3]=col[3].." "..col[1] end
  end
  return rows
end
```

2.3.6 Exact match control

When navigating in a table or executing sa.select function the highlighted line is the closest match. However when scanner is used a close match is not good enough so the default behavior in that case is that an exact match is required. It can be situations when an exact match is always required (or never required). This can be defined by cbTableMatch and cbSelectMatch callbacks. cbSelectMatch cannot be used for indexed list (4th argument to sa.select must be false) which means that it must be performed by a script since the built in source=”select” always use indexed list.

2.3.7 cbTableMatch

cbTableMatch is used to control how to handle requirements for exact match in Table.
See example below.

Retval=cbTableMatch (table, column, scanner)

Variable	Usage
table	Selected table
column	Column name
scanner	false: Scanner not used table: Scanner used
Retval	nil: Default behavior true: Exact match required false: Exact match not required

Ex: cbTableMatch

```
function(table, column, scanner)
  if table.name=="Shoe table" and column=="ID" then
    return true -- Exact match required
  end
  return nil
end
```

2.3.8 cbSelectMatch

cbSelectMatch is used to control how to handle requirements for exact match in sa.select.
See example below.

Retval=cbSelectMatch (format, field, prompt input, text, scanner)

Variable	Usage
format	Selected format
field	Field in format
prompt	Prompt used in sa.select
input	Text entered in search field
text	Text in the selected line
scanner	false if scanner not used table if scanner used
Retval	nil: Default behavior true: Exact match required false: Exact match not required
Note	Indexed list cannot be used. (4th argument in sa.select must be false)

Ex: cbSelectMatch

```
function(format, field, prompt, input, text, scanner)
  if format.name=="Price Demo" and field.name=="Marked by" then
     return true  -- Exact match required
  end
  return nil
end
```

2.3.9 cbTableSelect

Callback cbTableSelect can be used to modify the runtime parameters of a table. This can be used to hide columns from being shown or to change a columns starting input mode from Alpha-numeric to numeric input. It can also be used to control what policy decides which item is displayed at top when entering the table. It happens (is called) just before interacting with the user when selecting from table.

cbTableSelect(tTable)

Variable	Usage
tTable (input/output)	Contains properties of the selected table. Some: tTable.name – table name tTable.column – table with information about all columns indexed by integer and name. Set tTable.column[idx].keyformat = "%d" to force start in numeric mode. Set tTable.column[idx].hidden = true/"true" to prevent a column from being shown. Set tTable.column[idx].starttop = false/"false" to start showing item with id 1, the same way as in 40.00.01.02.
	This callback does not have any return values

2.3.10 cbInput

cbInput is called before sa.input is called. Retval controls weather input shall occur or not.

Retval=cbInput (format, field, prompt, value)

Variable	Usage
format	Selected format
field	Field in format
prompt	Prompt (string or table)
value	Actual value
Retval	true: No Input nil: Input
Note: sa.scanner(false) will disable the scanner. Original state will be restored automatically.	

Example: Prompt for value until > 0

```
function(format, field, prompt, value)
 if format.name=="Price Demo" and field.name=="WAS price" and (tonumber(value) or 0)>0 then
     return true
  end
  return nil
end
```

2.3.11 cbSelect

cbSelect is called before sa.select is called. Retval controls weather input shall occur or not.

Retval=cbSelect (format, field, prompt, value)

Variable	Usage
format	Selected format
field	Field in format
prompt	Prompt (string)
value	Actual value
Retval	true: No Input nil: Input

Example: Select until “Mats is” selected

```
function(format, field, prompt, value)
  if format.name=="Price Demo" and field.name=="Marked by" and value=="Mats" then
     return true
  end
  return nil
end
```

2.3.12 cbTableNoMatch

cbTableNoMatch is used to define if it is possible to return from table search without any match found when EN is pressed.

Retval=cbTableNoMatch (table,input)

Variable	Usage
table	Selected table
input	Input field
Retval	true: return nil: No return

Example: Accept empty return value if the text “Sato” is searched for

```
function(table,input)
  if table.name=="Shoe table"  and input=="Sato" then
     return true
  end
  return nil
end
```

2.3.13 cbFormatData

cbFormatData is used to execute pre- and postformat code.
The callback is located before and after the fields are executed.

cbFormatData (format,action,start,bCompleted,sErr)

Variable	Usage
format	Active format (=Format)
action	true: execute all fields. (Once for each batch.) false: execute only counters and related fields.
start	true: pre format false: post format
bCompleted	nil: not supported. false: format has been aborted (PU,MU or sa.abort) true: All fields generated successfully.
sErr	nil: no script error or not supported. ~nil: Lua script error string.
Note	If error in user defined (se.) callback the call looks like this: cbFormatData({},nil,nil,nil,<error string>) Usage: if format.name==nil then print(sErr) else <see below> end

Example:

```
function(format,action,start,bCompleted,sErr)
  if start then
    -- pre-format
  else
    -- post-format
    if sErr then
      -- cbFieldData,cbFormatData or a field script have generated an error
      --print(sErr)
    else
       if bCompleted then
        -- All fields generated successfully
      else
        -- Field generation aborted (by PU,MU or sa.abort)
      end
    end
  end
end
```

2.3.14 cbFieldData

cbFieldData is used to execute code before each field is executed.

cbFieldData (format,field,action)

Variable	Usage
format	Active format (=Format)
field	Active field (=Field)
action	true: execute all fields. (Once for each batch.) false: execute only counters and related fields.

Example: Make sure that the field "WAS price" does not use old value after Table selection (Row) has changed. (field.values[field.prompt] contains previous entry from sa.input.)

```
function(format, field, action)
  if action
    and format.name=="Price Demo"
    and field.name=="WAS price"
    and field.ID~=Row.id then
      field.ID=Row.id
      field.values[field.prompt]=nil
      field.value=nil
  end
end
```

2.3.15 cbSelectSort

cbSelectSort is used to change the sort function when selecting from sorted lists.

retval=cbSelectSort(id)

Variable	Usage
id	Must be of type string. “FMT” is reserved for select format menu “TBL” is reserved for select table menu
retval	The sort function for table.sort (see Ref.[3])
Note	The sort function can also be directly provided. See sa.select and sa.tableSelect

Below function is included in SA. It handles sorting of strings with leading numbers. Format 2.PriceDemo will be sorted before 10.PriceDemo. To use it, just put the string “FMT” or “TBL” as the 6th argument to sa.select.

```
function cbSelectSort(id)
  --* Execute once
  if not(cbSelectSortTbl) then
     --* return number or string
     local function Type(s)
       s=type(s)=="string" and s or ""
       local i,j=string.find(s,"^%d+")
       local n=i and tonumber(s:sub(i,j)) or s
       return n
     end

     --* sort function based on type
     local function Sort(a,b)
       a,b=Type(a),Type(b)
       if type(a)==type(b) then
         return a<b
      else
        return   string.lower(tostring(a))<string.lower(tostring(b))
       end
    end

    --* put id in table
    cbSelectSortTbl={FMT=Sort,TBL=Sort}
  end

--* return sort functions based on id
  return cbSelectSortTbl[id]
end
```

2.3.16 cbTranslate

cbTranslate is a callback that is called when the language changes and makes it possible to provide translations to the application. Example placed in Start script translates the name of the table and the title of side menu (F1).

```
--* Table with translations. For language codes see ISO 639-1.
--* [ "<original text>"]={<language code>="<text>",<language code>="<text>",<etc>},

local tr={
  ["QSR Demo"]={sv="Välj Maträtt", en_US="Select Dish"},
  ["sa.menu"]={sv="F1 "..sa.txt("sa.menu"),en_US="F1 Menu"},
}

function cbTranslate()
  local l=sa.language() or "en_US"
  for k,v in pairs(tr) do translate[k]=v[l] end
end
```

2.3.17 cbWelcome

cbWelcome is used to change the default welcome-text at startup.

```
--* Welcome message at startup
function cbWelcome()
  sa.msg({"Don’t Worry","Be Happy"},nil,1)
end
```

2.3.18 cbSdbTitle

When selecting in table, cbSdbTitle is used for providing a more informative title than the table name. The title is automatically translated. The function must return a string.

```
function cbSdbTitle(title,tableName)
  if tableName=="Shoe table" then
    return "Select Shoe Name"
  else
    return tableName
  end
end
```

2.3.19 cbCustomCommand

cbCustomCommand is used for providing the Android AEP application with an additional command string to send to the printer. It will be sent once for each print job, after all other fields, just before <Q1>.

retval=cbCustomCommand()

Variable	Usage
retval	The command string to be sent to the printer, as a Lua string.
Note	cbCustomCommand shouldn't have any side effects, as that would give different results in the Android AEP application compared to other platforms.

Example: To put an additional text field at position (100, 100) when printing on an SBPL printer, you could use the following function.

```
function()
  return "\027H100\027V100\027XUSome text"
end
```

2.3.20 cbLabelObject

cbLabelObject can be used to add extra render object(s) to a label before printing. This function is called in SA before the label is rendered.

cbCustomCommand(format, label)

Variable	Usage
format	The format in table format.
label	The label object.

Example: Print format parameters to console and add a circle object to the label

```
function(fmt, lbl)
  if resolver then return end -- for breaking loops/detecting in AEPWorks
  -- Print format
  print("json.encode(fmt)", json.encode(fmt))
  io.flush()
  -- Add circle to label
  local c = circleObject.new(200,200,100,4)
  lbl:add(c)
end
```

2.3.21 cbAtExit

cbAtExit is a callback that is called just before SA exits. It provides the opportunity, for a few seconds, to run some code before SA exits. After a few seconds SA is forcefully killed to make sure that SA really exits.

2.4 _return

With _return object callback it is possible to take control over what object is to be executed next. Add to each object

Retval=o._return(o,n)

Variable	Usage
o	Current object
n	Default next object
Retval	The callback returns the object where execution will continue nil: Default next object will be used
Note: This is an advanced operation that will be further explained in Application Notes STL00266.	

Ex. sa.objects.quantity._return=function(o,n) <add code> return n end

2.5 _f1

This is an object method to implement context sensitive side menu. If the object o. _f1 is present then o._f1 will replace side menu (sa.f1) temporary.

Note: This is an advanced operation that will be further explained in Application Notes STL00266.

2.6 nxt

nxt (se.nxt) is not a callback, it is the default function for format navigation.
A Format must be loaded and it sets which field to execute next after finishing current field.

Retval=nxt(field)

Variable	Usage
Field	For relative position use "+" or "-" immediately before the index "-2" For absolute position use a string or number "2" or 2 Field names can be used "Shoe name" If false (boolean) then format execution is ended after current field. Illegal values will be dismissed.
Retval	If field is nil nxt returns current field (Format.index)
Note1: The table Format.renderdata contains the result for all fields. Note2: #Format is the number of fields in format.	

Note3: This is an advanced operation that will be further explained in Application Notes STL00266.

2.7 Field:Size

Size is a field method that is used to calculate its size without rendering on label.
For return values see STB00011 fieldSize()

2.8 SA reserved table name

The table name **Translate** is reserved for translation file

The first column is the used tag and the rest is a language code.
This is a way to change existing or merge new words into the internal (SA) translation table.

tag	en	sv	da	de	es	fr	it	nl	no	en_US
Mark Down %	Mark Down %	Prissänkning %	Prissænkning %	Rabatt %	Rebaja %	Remise %	Sconto %	Afprijzen %	Prissenking %	Mark Down %
Marked by:	Marked by:	Märkt av:	Mærket af:	Marked by:	Marcado por:	Marqué par:	Etichett.Da:	Getek.door:	Merket av:	Marked by:
VAT %	VAT %	MOMS %	MOMS %	MwSt %	IVA %	REMISE %	IVA %	BTW %	MOMS %	VAT %
English	English	Swedish	Danish	German	Spanish	French	Italian	Dutch	Norwegian	English_US
ID	EN	SV	DA	DE	ES	FR	IT	EN	NO	en_US

3

SA API library

SA API library functions are defined in a separate namespace (sa.)

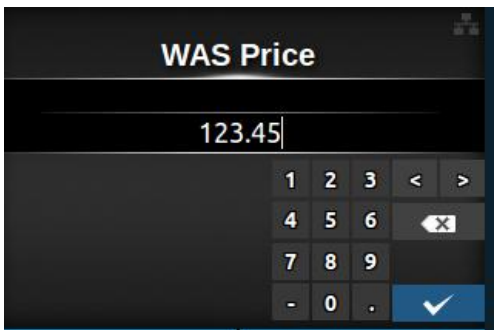
3.1 input

sa.input is designed for input of any type. It keeps track of latest input value.

Retval1, Retval2, Retval3 = sa.input (prompt, startval, inputformat, designvalue,new)

Variable	Usage
prompt	Title. (Translated internally.)
startvalue	This value is displayed first time after format selection. Next time previous entered value will be used.
inputformat	%length[.decimals] s|u|d|n|f (controls the keyboard mode) s=string, u=unsigned integer, d=signed integer, n=unsigned float, f=signed float (0 or empty length means infinite length) (0 or empty decimals on a float defaults to 2) Default is %s
designvalue	For use in design time (AEP Works)
new	true : Don't use previous value as default
Retval1	nil if key was an escape key (PU or MU) input value (string) if key was "EN"
Retval2	true : Scanner was used nil : Scanner was not used
Retval3	true : Keyboard was used nil : Keyboard was not used
Note 1: Scanner input is enabled inside this function. To disable scanner, see cbInput	
Note 2: Keyboard detection requires using sa.keyboardMatch() and sa.keyboard()	

Ex: sa.input ("WAS Price","","%3.2n","999.99")



3.2 select

sa.select is designed for list selection. It keeps track of latest input value.

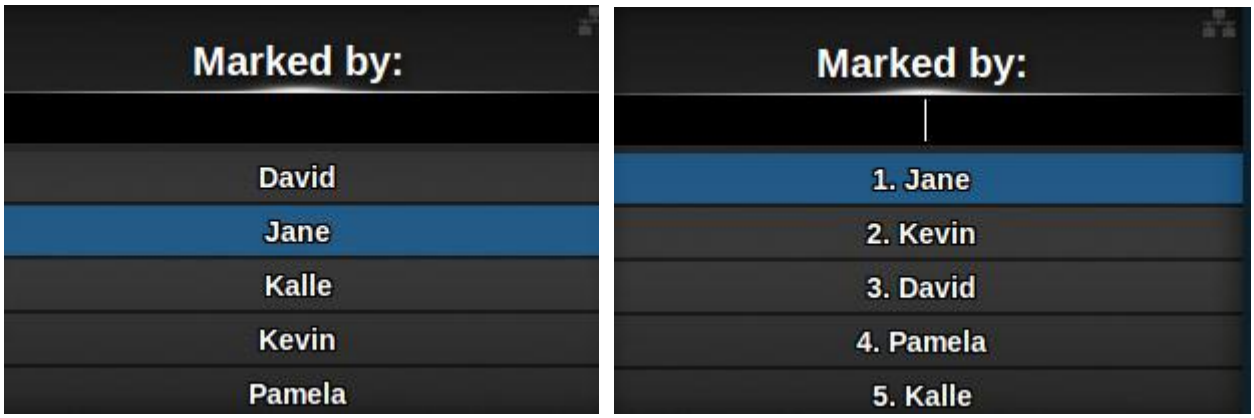
Retval1, Retval2, Retval3, Retval4, Retval5 = sa.select (prompt, table, index, indexed,plain,sort)

Variable	Usage
prompt	String to be displayed in first line. Translated internally.
table	Table or comma-separated list
index	Start index in list (default 1)
indexed	false. Use sorted list true. Use index (default)
plain	false. Accept value based on cbSelectMatch true. Accept any value.
sort	String. Call cbSelectSort to find sort function Function. Sort function according to table.sort (see Ref.[3]) nil. Default sort function
Retval1	Selected item
Retval2	Selected index
Retval3	Input text
Retval4	false: Scanner was not used true: Scanner was used
Retval5	false: External keyboard was not used true: External keyboard was used
Note 2: Keyboard detection requires using sa.keyboardMatch() and sa.keyboard()	

Example:

Left: sa.select ("Marked by:",{"Jane","Kevin","David","Pamela",”Kalle”},nil,false)

Right: sa.select ("Marked by:",{"Jane","Kevin","David","Pamela",”Kalle”})



3.3 key

sa.key(key) returns and manipulates key

key	Usage
nil	Returns last key
string	Set last key to string
true	Returns last key after object change This is a copy of key value when object changed. The value will remain until clear (false below) or set (string above).
false	Clear both above
Note: Formats are executed within the same object (formatData).	

Ex. Check if last key was Enter
if sa.key()==”EN” then …

3.4 keyRet

sa.keyRet returns true if the last key was "PU" or "MU"(long press “PU”)

Ex: Y=sa.input(txt("YEAR"),Y,"%4n") or Y
if sa.keyRet() then return end

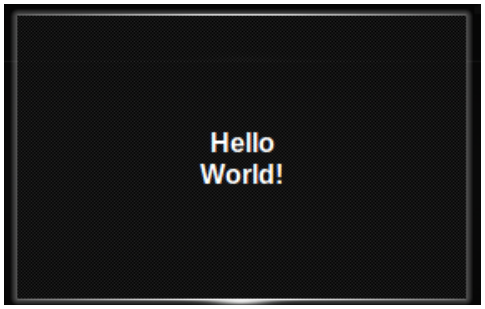
3.5 msg

sa.msg is used to display a message.

sa.msg (text,nil,timeout)

Variable	Usage
text	String or table (more than one row).
nil	nil (not used by CLNX)
timeout	Time to stay in display in seconds If nil then infinite
Default text is "Invalid". Translated internally. To leave press confirm button (F2).	

Ex: sa.msg({"Hello","World!"})



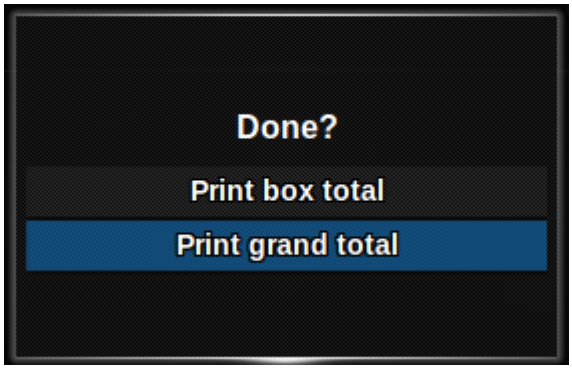
3.6 confirm

sa.confirm is used to inform the user that a decision must be confirmed.

Retval1,Retval2 = sa.confirm(text,options)

Variable	Usage
text	String
options	Table (list of options)
Retval1	true if confirm (F2), false if reject (F1)
Retval2	Selected option or nil
Note: Translated internally. Default text is "Delete?".	

Ex: local R1,R2=sa.confirm("Done?",{"Print box total","Print grand total"})



3.7 inputCheck

sa.inputCheck is designed for use in Formats and check validity of input (barcodes only)

Retval = sa.inputCheck(data, fieldtype, barcode)

Variable	Usage
data	String to check
fieldtype	Default is actual fieldtype
barcode	Default is actual barcode(type)
Retval	Boolean

Ex:
print(sa.inputCheck("1234567890123","barcode","ean13")) → false
print(sa.inputCheck("1234567890128","barcode","ean13")) → true

3.8 split

sa.split will split a string into a table. Pattern can be used.

Retval = sa.split(string, separator, patternmatch)

Variable	Usage
string	String with separators
separator	String indicating split position. Default is ","
patternmatch	Boolean, false or nil will disable pattern matching. Ex. If patternmatch is true then the separator "%a+" will split on every group of letters in string.
Retval	Table

3.9 quantity

sa.quantity set the number of labels to print and overrides the QUANTITY input.

It can also be used to configure other QUANTITY-related behaviors by passing in a table with arguments, e.g:

```
{
  list={1,2,5},
  loopQty=1,
  previewQty=2,
  qty=1,
  returnScreen="screenName", -- screen based printers only
  timeChanged=true, -- update time fields before printing

}
```

The table arguments passed in will update the options. Individual items are cleared by setting false. `previewQty=2` is used as preset QTY of 2 for preview screens. `loopQty=1` will enable a confirm dialog after print to enable printing an additional new item quickly. `list={1,2,5}` will show a list where the user can select between printing 1,2 or 5 labels.
`previewQty` and `qty` are automatically reset to nil after the print cycle.

sa.quantity(labels|opts)
sa.quantity(false)
sa.quantity(0)
quantity,options=sa.quantity()
option=sa.quantity(nil, "option")

The new implementation is not fully backwards compatible with the previous behavior if the labels argument is a table it will be interpreted as an opts argument. Otherwise as before, 0 can be set to print no labels. It will also skip the `loopQty`. Setting to false, will clear it. Passing other types of arguments to sa.quantity() to avoid printing labels is not recommended.

3.10 menuBase

sa.menuBase (abort)

Variable	Usage
abort	true : Must be set to a true value
Note	Menu Base is first child in menu (sa.menu[1]) object

3.11 scanner

sa.scanner is used to enable/disable scanner input or read actual state

Retval=sa.scanner()
sa.scanner(state)

When called without arguments the current state is returned.

Variable	Usage
state	true: enable false: disable nil: Retval

3.11.1 scannerMatch([pattern])

sa.scannerMatch() is used to control which USB keyboards/scanners that SA should control. The default behavior is to control devices that have "scanner" in their name.

Retval=sa.scannerMatch()

Return the current matching pattern (default "scanner") for USB-scanner.

sa.scannerMatch(pattern)

Variable	Usage
Pattern	A string or a table with strings that must be found somewhere in the device name. "" matches all USB keyboards.

3.11.2 scannerConf(conf)

sa.scannerConf() can be used to control some behaviors of how scanners are handled. There are two builtin configurations. For hid scanners (see sa.scannerMatch()), and for Bluetooth scanners, but more ports can act as scanners by adding them to conf when they exist in system.ahd().

```
{
  hid={ timeout=0.1, cp="UTF-8", fs=13, enable=true},
  Bluetooth = { timeout=2, cp="UTF-8", fs="\r\n", fn=lua_function, enable=true }
}
```

For hid only `timeout` can be modified.
The fn-parameter is preloaded with a builtin lua function that searches for the field separator `fs`, and then it converts it to internal scanner data using sa.dataConv(). The lua_function can be overloaded. The prototype is

```
function (port,t)
  -- port: is the port name from autohuntd.
  -- t[1]:inout argument is the data from the scanner (complete/incomplete)
  -- t[2]:out argument is the data after conversion
end
```

When the function is called, only `t[1]` exists. When function deems the scan packet as completed, it sets the converted packet data in `t[2]`. The data in `t[1]` that produced `t[2]`, should be removed from `t[1]`. The data format in t[2] must match the format from sa.dataConv().

Example to add TCP Port1 as scanner (ahd-port name 1024)

```
local conf=sa.scannerConf()
conf["1024"]=conf.Bluetooth  -- use the Bluetooth conf table for 1024
conf.Bluetooth=nil           -- to make sure not to modify Bluetooth
sa.scannerMatch(conf)        -- activate it
```

That can be used to emulate scanner behavior by connecting to port 1024.

Retval=sa.scannerConf()

Return the current configuration. Use sa.scannerConf(conf) to update.

sa.scannerConf(conf)

Updates the configuration with the data in conf. To delete a port, set it to false. NB! The `hid` and `Bluetooth` port configurations cannot be deleted.

3.11.3 dataConv(port,data)

sa.dataConv() is a utility function to convert the scanner data into the internal format. It is used by the builtin function to produce t[2]. The port-parameter looks up the from codepage `cp` to "UTF-8", and then converts the string data into the internal format and returns it as a string.

3.12 keyboard

sa.keyboard is used to enable/disable input from USB keyboard or to read actual state

Retval=sa.keyboard()
sa.keyboard(state)

When called without arguments the current state is returned.

Variable	Usage
state	true: enable false: disable nil: Retval

3.12.1 keyboardMatch([pattern])

sa.keyboardMatch() is used to control which USB keyboards that SA should control. It is called with the unclaimed left-overs after sa.scannerMatch(). The default behavior is to control the keyboard in the browser.

Retval=sa.keyboardMatch()

Return the current matching pattern for USB-keyboard.

sa.keyboardMatch(pattern)

Variable	Usage
pattern	A string or a table with strings that must be found somewhere in the device name. "" matches the remaining USB keyboards not claimed by sa.scannerMatch()

3.13 timeOffset

sa.timeOffset() is the value in seconds that will be added to RTC when a time-source is requested. If flag is omitted, nil or true timeChanged flag will be set to true. Retval is actual timeOffset.

Retval=sa.timeOffset(seconds,flag)

Ex. Make time fields in all formats return RTC+1h
sa.timeOffset(3600)

3.14 timeChanged

sa.timeChanged is a flag that trigger a recalculation of time fields (and fields that copy from them) before printing. The flag is cleared.

Ex. Recalculate time fields next printing
sa.timeChanged=true

Note: When Quantity is requested then the format is ready for printing. If printing does not occur within 60 sec. all time fields will be recalculated automatically.

Note: sa.quantity({timeChanged=true}) can done in Start, to set sa.timeChanged at each Quantity-prompt.

3.15 txt

Translates text or table according to internal translation files using configured language.

Retval=sa.txt (text)

Variable	Usage
Text	String or table to be translated. (Case sensitive)
Retval	Translated text or table Search order: 1. Use SA translation table (table:translate) 2. Use f/w translation table (function:config.xlate see STB00011) 3. Use the tag.
Note: **sa.language**() returns the configured language code. (ISO 639-1)	

3.16 Formatters

Formatters are used in data pipe process to format data according to type and locale but are also usable in other places.

Name	sa API	Base	Description
Default	sa.printf(format,value)	string.format(format,value)	Lua ref.
Date&Time	sa.ldate(format,value	system.ldateFormat(format,value)	STB00011
Currency(sign)	sa.lmon(value)	system.lmonFormat(value)	STB00011
Currency(text)	sa.limon(value)	system.limonFormat(value)	STB00011
Number	sa.lnum(value)	system.lnumFormat(value)	STB00011

Ex.
sa.limon (123) → USD 123.00
sa.ldate ("%x", os.time()) → 4/23/2016
sa.printf("%03xH",100) → 064H

3.17 trim

sa.trim trims string whitespace (beginning and end)

Retval=sa.trim(string)

3.18 tableSelect

sa.tableSelect is based on sa.select with the list provided from a database table.

Retval1, Retval2, Retval3, Retval4=sa.tableSelect(prompt,column,tableName,text,indexed,sort)

Variable	Usage
prompt	String to be displayed in first line. (Search field must also fit into the first line. See below) Translated internally.
column	Column name in database table
tableName	Name of the database table
Text	Default selection
indexed	false. Use sorted list and search field true. Use index (default) If # of items > 9 then search field will appear.
Sort	string. Call cbSelectSort to find sort function function. Use function to sort list nil. Default sort function
Retval1	Selected text
Retval2	Selected index
Retval3	Input text
Retval4	false: Scanner was not used table: Scanner was used
Note 1: Indexed view will use numeric input mode, Not Indexed will suggest numeric if any item starts with a number Note 2: List items are limited to 10000 Note 4: The table sa.pk[<tablename>] will contain the latest selected primary key Note 5: Callback cbSelect is applicable	

3.19 tableMatchRow

sa.tableMatchRow is used to query for one and only one match in a sdb table

row=sa.tableMatchRow(t,query,column)

Variable	Usage
T	Loaded table or table name
Query	Query for column
column	For loaded table default column is t.display (column=nil) Column name or number If column has legal value it is used otherwise default is 1
Row	nil: not found table: array (indexed) and column names with value
Note:	See STL00266 and WSP01141

3.20 tableMatchRows

sa.tableMatchRows is used to query a sdb table

rows=sa.tableMatchRows(t,query,column,n,offset)

Variable	Usage
T	Loaded table or table name
Query	Query for column
column	Column by name or number (numeric) Default column is t.display or 1
N	Max number of rows to fetch Not a number: max 1000
Offset	Offset in selection Default: 0
Rows	nil: table not found table: array of rows and rows.columns and rows.info (notes below)
Note1:	rows.columns is a table with xref for columns with index=name and name=index (Used by sa.makeRow below)
Note2:	rows.info contains info about selected table lastMatch offset to last match in selection columnIndex index for used column sort how the data is sorted query input search string column name of used column format format for data rows delivered number of rows maxrows total number of rows in table For further details see STB00011 f/w API spec

3.21 makeRow

sa.makeRow creates the global Row variable that is used for table selection

row=sa.makeRow(rows,columns,n)

Variable	Usage
Rows	table: if rows[n] is a table then use rows[n] not table: return empty table
columns	table: xref, see note for sa.tableMatchRows nil: use rows.columns
N	See above
Row	Return value (same as Row, created in function)

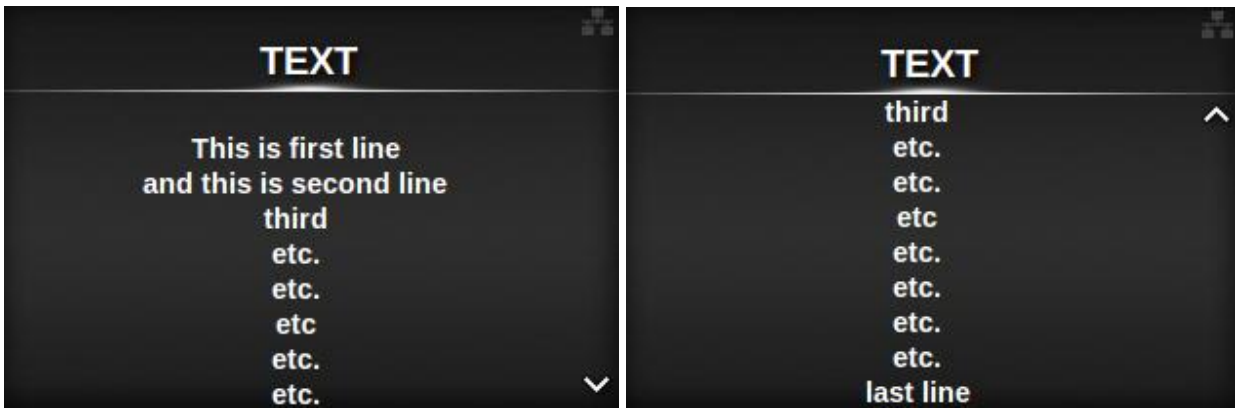
3.22 displayText

sa.displayText is used to display and vertical scroll a text on several lines.

sa.displayText (title,text)

Variable	Usage
Title	The title in display
Text	Text body with newline ("\n") as line separator.
Note: Not compatible with displayText for TH2 printer	

sa.displayText("TEXT","This is first line\nand this is second line\nthird\netc.\netc.\netc\netc.\netc.\nlast line")



3.23 checkDate

sa.checkDate checks legal dates between 1970-01-02 00:00:00 – 2037-12-31 23:59:59

Retval=checkDate(Year,Month,Day,Hour,Minute,Second)

Variable	Usage
Year	1970-2037 Default 1970 If 00-37 then 2000-2037 is assumed
Month	Default 1
Day	Default 1
Hour	Default 0
Minute	Default 0
Second	Default 0
Retval	Boolean

Example:
print (sa.checkDate(2015,2,29)) → false
print (sa.checkDate(2016,2,29)) → true

3.24 inputDate

Input dates between 1970-01-02 – 2037-12-31

Retval=sa.inputDate(Year,Month,Day)

Variable	Usage
Year	Default now If 000-037 then 2000-2037 is assumed
Month	Default now
Day	Default now
Retval	The date expressed as seconds from epoch (1970-01-01 00:00:00)

Example:
print (sa.ldate ("%Y-%m-%d", sa.inputDate())) → 2009-10-21

3.25 inputTime

Input hours and minutes

Retval=sa.inputTime(Hours,Minutes)

Variable	Usage
Hours	Default now
Minutes	Default now
Retval	Hours and minutes expressed as seconds

3.26 displayHTML

Use HTML tags to format display content.

Retval1,Retval2=sa.displayHTML(content,prompt)

Variable	Usage
content	string: HTML string table: (for advanced use)
prompt	The title If exists it will override prompt tag if content is a table
Retval1	Key pressed
Retval2	Return table (for advanced use)

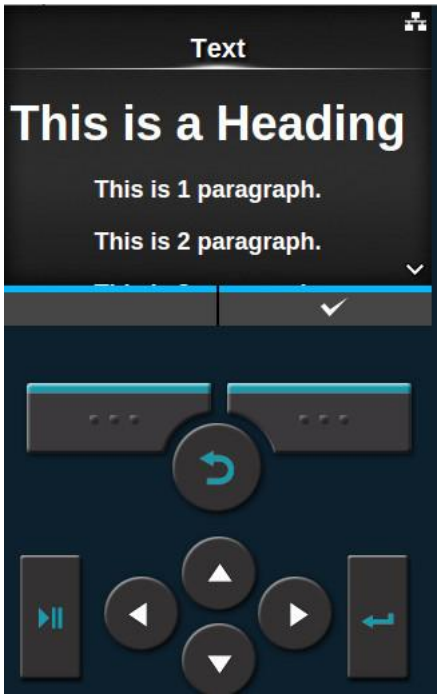
Example left:

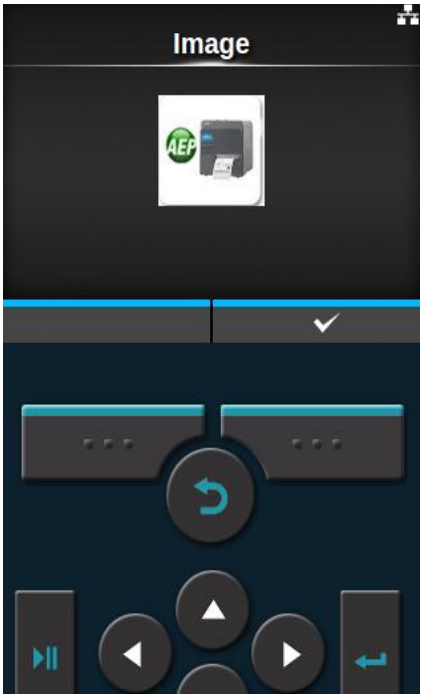
```
r=sa.displayHTML("<!DOCTYPE html><html><head><title>Page Title</title></head>
<body> <h1>This is a Heading</h1><p>This is 1 paragraph.</p><p>This is 2 paragraph.</p>
<p>This is 3 paragraph.</p> </body></html>","Text")
```

Example right:

```
local I="<img src='base64 encoded image'>"
r=sa.displayHTML(I,"Image")
```

(For base64 encoding: https://www.base64-image.de)





3.27 initUTC

This function is targeted for PW2NX, to synchronize the system clock with the UTC time. It requires WLAN to work. The time zone can be specified using settings, or by passing it as argument, e.g. sa.initUTC({zoneinfo="Europe/Gothenburg"})

4

Power fail storage (Pfs)

Pfs is an easy way to store data nonvolatile in flash, however all data is written each time any data is modified. This means that it is intended for small amount of data like totals, counters and queues. For logging purposes, file io with append is recommended instead. Note: Intensive writing to flash is not recommended. A solution with ram should be considered.

Pfs (or se.Pfs in global space) is a repository where access to the variable interacts with the file system to ensure that the information is non-volatile. The implementation is a so called funcTable, a table that also have functions behavior.

Writing and reading from Pfs is straight forward table syntax
Pfs["A"]=25 or Pfs.A=25
Pfs.A=Pfs.A + 10

Any type of data can be stored
Pfs.B = {"One", "Two", "Three", n=3}
print (Pfs.B[2]) → "Two"
print (Pfs.B.n) → 3

Table methods for the Pfs table cannot be indexed; they must be transferred as argument to the function. See below. For indexed Pfs tables (like Pfs.B above) standard Lua is used. However in this case, a write to file must be triggered separately. See Note below.

Method	**Usage**
nil	Pfs() (clear all)
#	Retval=Pfs("#")
copy	Retval=Pfs("copy")
insert	Pfs("insert,"[pos,] value)
remove	Pfs("remove"[, pos])
sort	Pfs("sort"[,comp])
getn	Retval=Pfs("getn")
maxn	Retval=Pfs("maxn")
concat	Retval=Pfs("concat"[, sep[, i[, j]]])
pairs	for i,v in Pfs("pairs") do … end
ipairs	for i,v in Pfs("ipairs") do … end

The storage is located in the file /ffs/apps/sa/pfs and a removal of that file will clear all data.
Note: Pfs use the index metamethods. This means that above example is not the same as
Pfs.B={} Pfs.B[1]=”One” Pfs.B[2]=”Two” Pfs.B[3]=”Three”

since Pfs __index metamethod is not triggered. The data will therefore not be stored to file immediately. However any instruction that trigger __index for Pfs will fix the problem.
For example: Pfs.B=Pfs.B

For more details about table methods see Lua Reference Manual.

5

SA Objects

SA objects are based on a generic object definition. The name of basic methods and attributes starts with underscore while SA specific methods and attributes start with lowercase.
Developers can add their own methods and attributes to objects and use the same name convention that is used in SA namely start with an uppercase letter.

5.1 Menu structure

SA consists of objects organized in a menu tree. See below.

Main menu is not shown because there is only one child, the object app.
If another child is added to main then a selection will be prompted.

In object app selectable tables and formats are combined into one list for selection.
Object app has 2 children formatTable and tableFormat that are called dependent of it was a format- or table based application that was selected.
sa.object.app.mode = 1/2 format/table mode
sa.object.app.name = selected format/table name

Both ways ends up in formatData where the format is executed. Finally quantity is called where number of labels in batch is requested and printing occurs.

main
.app
..formatTable
...formatData
....quantity
..tableFormat
...formatData
....quantity

Meny key (F1) has 4 children that are imported from file fnc.

f1
.fnc.print
.fnc.profile
.fnc.timeoffset
.fnc.info

5.2 Menu Objects

Above tree view shows that the menu consists of 6 object.

Name	Purpose
main	Root object. If it only has one child, call that child.
app	Start object. Define the globals Format or Table dependent on format/table based application.
formatTable	Define Row in Table. (Format based application. Format is selected and defines Table)
tableFormat	Find Format and define Row. (Table based application. Table is selected and defines Format)
formatData	Execute Format
quantity	Input quantity and Print batch

5.3 What is an SA object?

SA objects are Lua tables (functable) with a few defined attributes and methods.

_name, _title and _call are always defined by the creator. If _call is not defined then the default method _menu is called and the objects children are shown (_title) in display for selection. Object f1 is an example. Indexed attributes (1, 2, 3..) are reserved for children.

To create a SA object the function sa.new is used with name, title and function as argument.

sa:new("MyObj", "My First Object", MyFunction) -- name must be unique

The new object is returned and can also be found in sa.objects.MyObj.

5.4 Methods

When a SA object is created it also inherits methods.

5.4.1 _insert, _remove

These methods works in the same way as Lua table.insert and table.remove.

How to add an object to f1 menu in first position.
Create the function.
function MyFunction() sa.msg("My Testing",nul,1) end

Create the object.
local o= sa:new("MyObj", "My First SA Object", MyFunction)

Insert object in f1 menu, first position. (sa.f1 points to sa.objects.f1)
sa.f1:_insert(o, 1) -- put an object first in f1

How to remove the same object from f1. (The object still remains in sa.objects)
sa.f1:_remove(sa.f1[1])

See also: AEP Hub - APP00005 Add2F1 function (TH2 compatible).

5.5 Movement and customization

For minor customization, it is simple to override the _call function.
One example is to take control over the quantity object. The function asks for the number of labels to print and then print them.

```
local qc=sa.objects.quantity._call      -- save a pointer to original function
function sa.objects.quantity._call(…)  -- create the new function
  --* add some code

  local retval=qc(…)    --* call the original function and save the result

 --* add some more code

  return retval              --* return retval (or another object to execute next)
end
```

When an object returns nil then SA will check the stack trace and move to the topmost object, however if the object returns a pointer to another object then execution will continue in that object.

Where to move next is prepared by inserting objects as children. By replacing these children it is possible to customize the execution path. (For existing children see above Menu structure.)

It is also possible to take full control by adding the callback _return(a,b) to an object. See above.

6

System objects

6.1 Introduction

It is possible to access system objects to perform filtering of in-data and out-data, and to hook actions to events. This section describes how to access those objects. As SA is a single-threaded application, it is necessary to write the handles so that they run their code swiftly and then return, to prevent a sluggish system.

6.2 sa.events

This is a system events object containing the data handles (file descriptors, fd). SA registers these named I/O-channels:
`"online:io"` - this is the device handle of type deviceObject to read from to filter in-data. This is also the handle to write responses to.
`"online:out"` - this is the device handle of type deviceObject to read the responses from the parser (e.g. SBPL). To send it back to the host, use the `"online:io"`-handle.
`"online:in"` - this is the device handle of type deviceObject to write the filtered in-data to. It will then be parsed by the current emulation parser.
`"events"` - this is the event handle for the system events enumerated in the text-file `"/rom/autoload/evt.lua"`. This is used with system.callbacks(), see next section.
`"gui"` - this is the gui handle for the CLxNX GUI.
`"scanner"` - this is the device handle name used for scanner. NB! There may be many, and they all have the same name. The device handles are of type deviceObject.

Please note that `sa.events:init()` will be issued when a scanner is plugged in/out and after errors, and that will remove your changes if you don't design for it. See the below example for how to handle init and how to run a filter function in front of the default.

```
local _init = sa.events.init
sa.events.init=function(t)
  _init(t)
  -- configure my changes
  local origFn
  local fd=sa.events:get("online:io")
  origFn = sa.events:replace(fd,fd,function(fd) print("this runs before origFn")
return origFn(fd) end)
end
```

Read more about `deviceObject, system.ahd(), system.newEvents(), gui` in STB00011. Another source of information is WSP01126 in AEP Hub.

6.3 sa.conf.callbacks

This is the system callbacks object used by SA to hook into the notification events. Read more about system.callbacks() in STB00011. SA listens to the `aep`,`gui`,`ntagi2c` and `statusd` event groups. To get a listing of those events, see the contents of `/rom/autoload/evt.lua`. To refer to an event in statusd, e.g. status, the design pattern is `require("autoload.evt").statusd.status`.

6.4 Writing HTML to be displayed in the GUI (browser)

For scripts such as AEPService/Upgrade that displays a progress bar, CLxNX-specific API:s can be used. This is described in STB00011, but the kickstart guide is here.
1. Create your own GUI connection

```
myGui = require("autoload.gui").new()
myGui:run(sa.conf.callbacks)
```

2. Write your own HTML message

```
myGui:show({type="html",content="Hello world"})
```

3. Get response from the message

```
local r = myGui:receive()
dprint("I got this response:",json.encode(r))
```

The input functions used by SA (`sa.input(...), sa.select(...)`, ...) are implemented with the message types described in STB00011.

It can also be done like this:

```
local myGui=sa.events:get("gui")
myGui:show({type="html",content="Hello world"})
local r = myGui:receive()
dprint("I got this response:",json.encode(r))
```

6.5 The main loop of SA

As explained in STB00011, the CLxNX printer's LCD-process operates in AEP-mode when SA requests user input, and the gui handle switches to the other operating states depending on the operating conditions. The main loop can be described with these lines:

```
while true do
  -- part 1
  sa.events:resume(gui)
  while (not printing via AEP) do
    sa.events:run()
  end
  sa.events:suspend(gui)
  -- part 2
  while (printing via AEP) do
```

```
    sa.conf.callbacks.eventWait() -- symbolic code
  end
end
```

When the user inputs the data requested by the format including the QTY, part 1 runs. When the user is done inputting the QTY, part 2 takes on and the printer runs there until finished.

The typical filtering application runs only in part 1, and it will react to events (notification events, data input events) happening on the file descriptors in the sa.events's objects table.

7

Migration from TH2

7.1 XML (SA contract with AEP Works)

Defined in STB00102 for TH2 and in STL00255 for CLNX is compatible. It means that formats, tables, functions and fnc (F1) are compatible on a XML-level. Fnc for TH2 have more items and some of them are not compatible.

7.2 H/W Platform differences (STB00011)

The main difference comes out of modified keyboard and display.
display.*, keyboard.* does not exist and its handling is not performed by SA.

7.3 SA API (SA Standard Library)

Is mostly compatible. Some exceptions:

sa.confirm is based on a separate gui and has more options and special keys in CLNX.

sa.displayText was more advanced designed in TH2 with both keyboard, timeout and display options. CLNX implementation scroll text that is preformatted with newlines.
Argument order is different. TH2 did not have a title and separator could be defined.

sa.input input position always the same. In TH2 the prompt could use more than one line and "push" the input position down. In CLNX the prompt is on one separate line and input on another separate line. (Soft keyboard needs space.)

7.4 Menu

Settings for Application and Printer removed. (Edit, F1, MODE). Printer settings are accessed in system menu.

7.5 Mode

Mode removed from menu (for simplicity).
Format/Table mode could earlier hide incorrect associations between formats and tables.
Now all selectable tables and formats (applications) are shown in one list.
Change associations in AEP Works to get rid of this problem.

7.6 Menu objects

The menu is built on other objects and overriding of internal functions like menu.selectFmt, menu.selectTbl is not compatible.

7.7 Modify F1 in runtime

TH2 methods not compatible. How to do is described in this document, see 2.5 and 5.4.1 _insert, _remove.
See also: AEP Hub - APP00005, Add2F1 function (TH2 compatible).

8

Application Notes

Application Notes for TH2 (STB00257) are still applicable.
Application Notes for CLxNX are found in STL00266.

9

Document

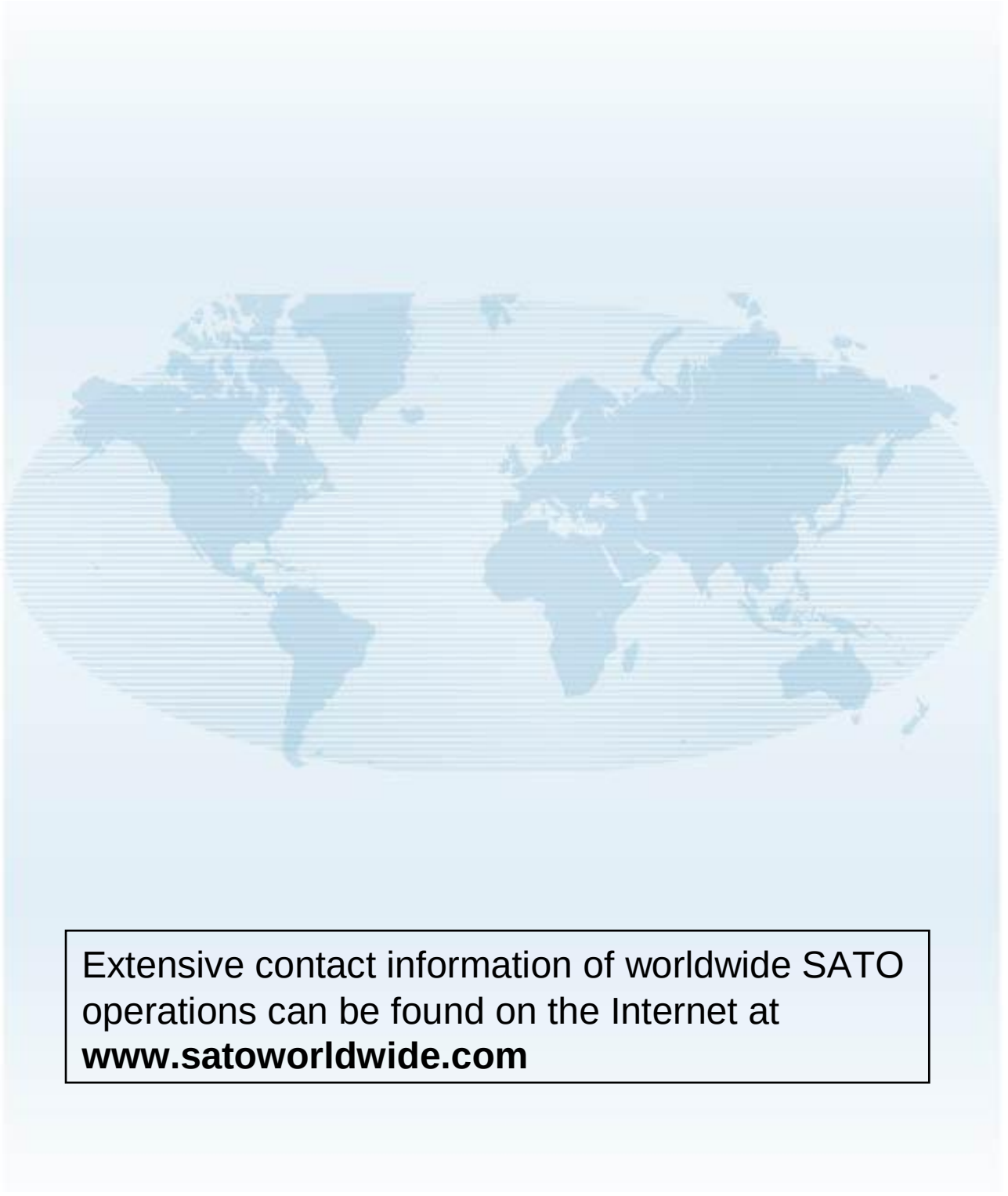
9.1 References

[1] Lua Technical Firmware API, STB00011.
[2] Standard Application XML, STL00255.
[3] Lua Reference Manual

9.2 Revision history

Revision	Name	Comment	Date
PA1	Staffan Gribel	First draft. Scanner not implemented yet. *Text in Italic*.	2015-09-08
PA2	Mats Hedberg	Changed front page	2015-09-17
PA3	Staffan Gribel	Added inputDate, inputTime, dateCheck, cbMenuTable, nxt and displayText.	2015-09-25
PA4	Staffan Gribel	Scanner is implemented	2015-10-02
PA5	Staffan Gribel	cbPredict removed, Field:Size added, objects more explaned	2015-11-03
PA6	Martin Dahlberg	Added descriptions of sa.events and sa.conf.callbacks	2016-01-21
PA7	Staffan Gribel	Modified cbSelectSort	2016-02-11
PA8	Per Andersson	Added callback cbLabelObject.	2016-02-16
PA9	Staffan Gribel	Added function sa.tableMatchRow	2016-02-23
PA10	Martin Dahlberg	Updated information in chapter 5	2016-03-01
PA11	Staffan Gribel	Created chapter 6 TH2 migration	2016-03-01
PA12	Lars-Åke Berg	Updated sa.events.init override example.	2016-03-10
PA13	Staffan Gribel	Reworked the entire spec. Added Pfs.	2016-04-10
PA14	Martin Dahlberg	Added descriptions of sa.scannerMatch, sa.keyboardMatch, sa.keyboard and updated sa.input, sa.select with the new additional return value	2016-04-13
PA15	Staffan Gribel	Updated Object movement and customization Added sa.menuBase and _return callback	2016-04-14
PA16	Staffan Gribel	Uppdated according to Peter Ekbergs request Added context sensitive side menu _f1	2016-05
PA17	Staffan Gribel	Added sa.tableMatchRows and sa.makeRow	2016-10-17
PA18	Staffan Gribel	Group of Labels in cbBatchDone	2017-03-08
PA19	Peter Ekberg Magnus Wibeck	Added anchor points. Minor language and formatting corrections.	2017-06-07
PA20	Staffan Gribel	Rows and Formats are reserved names	2017-08-28
PA21	Staffan Gribel	sa.displayHTML function Bookmarks for callbacks and other	2017-11-16
PA22	Martin Dahlberg	Added new features to sa.quantity()	2017-11-22
PA23	Martin Dahlberg	Updated the sa.quantity() description	2017-11-28
PA24	Staffan Gribel	Bookmarked displayHTML	2017-12-03
PA25	Lars Persson	Added callback cbAtExit.	2017-12-22

PB1	Martin Dahlberg	Added sa.initUTC()	2018-04-19



Extensive contact information of worldwide SATO operations can be found on the Internet at **www.satoworldwide.com**

SATO