MICS 6 Digital Data Collection System

Draft as of February 1, 2017

Developer's Guide

Table of Contents

Table of Contents

Overview

The sixth round of MICS introduces a new software platform for digital data collection using mobile devices known as Computer Assisted Personal Interview (CAPI). The new system, although it is still reliant on Census and Survey Processing System (CSPro) as the main data processing software, implements a range of new external components, exposing additional functionality not natively supported by CSPro. These external components communicate with CSPro applications through a Process Container software framework. Process Container is an application specifically developed to organize, categorize and manage functionality external to the CSPro application, while providing a consistent way of accessing the said functionality.

Even though the overall system design feels similar to the previous MICS phases, the underlying software architecture has been completely reworked around the Process Container framework. The data collection infrastructure still relies on hierarchical relations between the three data management systems – **menus**. These menus represent three tiers of the data collection organization: central office, supervisor, and interviewer, and are designed to drive the data collection process though invoking a series of data entry and batch edit CSPro applications.

In the previous system the external functionality was implemented in a form of separate unrelated utilities that were called independently from within the menu applications. The current system, however, uses process container environment as a standardized way of accessing external functionality from within CSPro applications.

Standard Project Folder Structure

Standard project folder contains all standard MICS CAPI applications and utilities as well as tools needed to prepare the static reference data needed for CAPI field operation. The project root folder contains the following subfolders:

- **ACCEPTED** – data for clusters accepted at central office
- **APPLICATIONS** – all CSPro applications relevant to CAPI data collection (see more details below)
- **ARRIVED** – data for clusters closed in field but not yet accepted at central office
- **CLOSED** – data for clusters closed by team supervisor
- **DATA** – clusters data on interviewer CAPI device
- **DELEGATED** – temporary data files for delegated interviews
- **FINALIZED** – data for clusters finalized by central office
- **PREPARATION** – tools and utilities needed for preparing survey reference data
- **RECEIVE** – data received by supervisor after synchronization via Bluetooth between supervisor and interviewers
- **REFERENCE** – survey static reference data, task template files, icons and CSS files
- **SYNC** – Bluetooth synchronization cache
- **TASKS** – tasks data files
- **TEMP** – temporary system files
- **UPDATE** – system updates folder
- **UTILS** – process container environment executables and other utilities
- **WORK** – working files

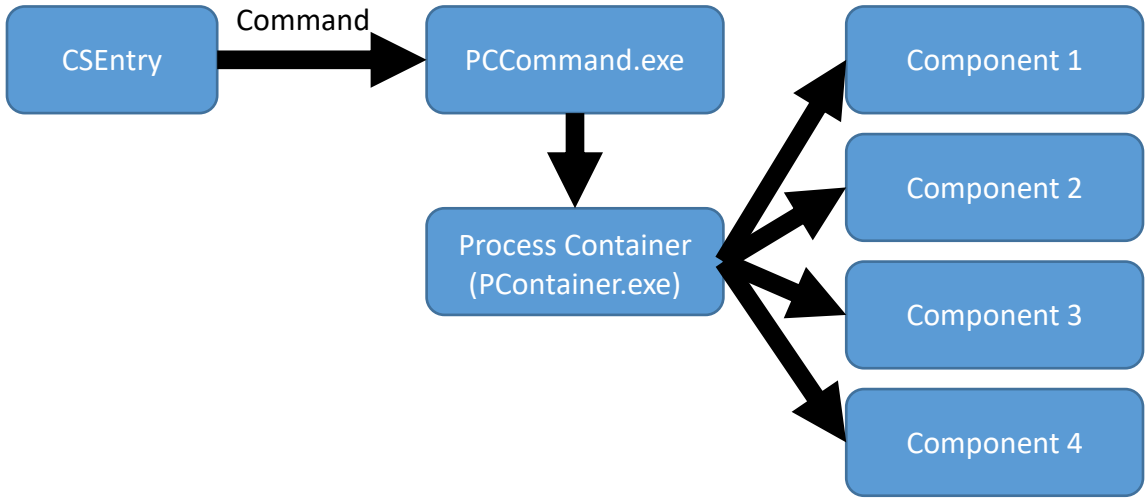
The **APPLICATIONS** folder is used to store all CSPro applications needed to run the CAPI system on all hierarchical layers: interviewer, supervisor and central office. The folder contains the following subfolders and applications:

- **CENTRAL**
    - **CentralMenu.ent** – main data management application for central office system. (This application is used to provide data management and progress reporting tools to survey technical staff working at the central location as well as to finalize data coming from the field.)
    - **StatusReport.bch** – batch application to tabulate survey data collection operation status report
- **COMMON** – reusable application modules and message files
- **DICTS** – CSPro data dictionaries for all applications
- **ENTRY** – CSPro entry applications for all survey questionnaires
- **EXPORT** – data export applications
- **INTERVIEWER**
    - **InterviewerMenu.ent** – main data management application for interviewer system
    - **UpdateTasks.bch** – batch application to dynamically update interviewer tasks based on completion of data collection in cluster
- **SUPERVISOR**
    - **SupervisorMenu.ent** – main data management application for supervisor system
    - **CKID.ent** – Check ID application to check structural integrity of collected data and provide data collection progress in cluster
    - **GenReviewTasks.bch** – application that generates tasks for questionnaire review by supervisor based on data collected by interviewer

Process Container

Process container is a Windows application environment designed to encapsulate a wide range of software functionality and expose it to other running applications such as CSEntry. Process container houses different components that serve as repositories for different functions, interfaces and data structures. These components can be invoked by the process container application through specialized command syntax. Commands are delivered to the process container environment through PCCommand.exe, separate child process. External applications such as CSEntry can dispatch the command using the PCCommand.exe process to the process container environment running on the background. The command is communicated to the environment and the environment processes it by passing it over to the corresponding component, which in turn executes the appropriate function and returns back to the environment. The command container environment then returns the command result to the PCCommand.exe process, which then immediately terminates itself, thus letting the external application know the command has been executed. **Figure 1.1** shows the described relationship between the process container environment, PCComand.exe process and an external CSEntry process.

Figure 1.1 Relationship external application (CSEntry) and process container environment



The process container environment process is represented by a PContainer.exe executable file. When run, process container initializes all of its components and switches to background mode. In this mode the environment detects instances of the PCCommand.exe process and is ready to execute all incoming commands.

Note: If the PCCommand.exe process is executed while the process container environment is not running on the background, the PCCommand.exe process will start the background instance of the process container automatically, assuming both executable files, PCCommand.exe and PContainer.exe are located in the same folder.

Command Syntax

The PCCommand.exe process expects a single command line argument, which represents the command to be relayed to the process container environment. The command itself is a string formatted as Universal Resource Identifier (URI). **Figure 1.2** describes command general syntax.

Figure 1.2 PCCommand.exe command syntax

COMMAND?ARG1=VAL1&ARG1=VAL1&ARG1=VAL1

The command consists of a command unique name and optional string of command arguments separated by a **"?"** symbol. A string of arguments may contain one or more key/value pairs separated by an **"&"** symbol. Below are some of the examples of existing commands:

- **GetBtAddress**
- **SetTempFolder?path=c:\temp**
- **SetLabel?name=lab1&content=Hello World!!!**

Upon receiving a command, the process container environment uses the command name to identify the appropriate component responsible for command execution. The environment then instructs the component to execute the function associated with the given command and supplies optional arguments where needed.

Components

The process container environment is responsible for parsing and routing command execution to appropriate external components. The actual functionality necessary to execute a command is stored inside the components themselves. There are seven main components currently implemented in the current version of the system:

1. Common commands
2. Environment variable space
3. Task list
4. Bluetooth synchronization
5. HTML report factory
6. Web browser
7. Data grid

Each component stores external functionality otherwise unavailable in CSPro. Through specially constructed process container commands, the environment invokes this functionality and exposes it to external applications such as CSPro entry and batch applications. The complete list of commands with descriptions is provided in **Appendix I** (p. 31-49).

Common Commands

The common commands component contains commands used for non-specific tasks such as process management, file management, environment management and debugging. There are also commands for displaying custom error messages and Windows notification balloons as well as setting global environment temporary folder parameter.

Environment Variable Space

Environment variable space (ENVI) is a component used to store temporary custom values accessible while the environment instance is active. ENVI is similar to CSPro working declared in global scope of entry and batch applications. ENVI variable values can be set and retrieved through commands from within CSPro applications. Although similar to CSPro working variables, ENVI variables provide a wider range of accessibility and functionality. The main advantage of an ENVI variable is that once its value is set it is accessible by any component of the entire system and is not tied to just one application. For example, an ENVI variable can be set in one CSPro application and then its value can be accesses from within another CSPro application as long as the environment instance is still active. Thus, ENVI provides a consistent way of storing temporary values shared between different parts of the entire environment.

ENVI provides four basic data types as follows:

1. **Int** – 32 bit signed integer values (-2,147,483,648 to +2,147,483,647, default value: 0)
2. **Float** – 32 bit floating-point values (-3.4×10^{38} to $+3.4 \times 10^{38}$, default value: 0)
3. **String** – string values of variable length. Literal strings are enclosed in single quotes. Default value: empty string.
4. **Bool** – Boolean values (true or false). Default value: true.

ENVI syntax is an LL(k) formatted string, consisting of one or more ENVI expressions separated by semicolon. For example: ***int** i = 10**; int** j = i + 5**;***

The above ENVI syntax contains two expressions declaring variables ‘i’ and ‘j’, setting values 10 and 15, respectively.

Two environment commands are used to submit ENVI syntax to the environment and retrieve ENVI values:

- **ExecEnvi** – is used to parse and execute ENVI syntax, and accepts single argument commands containing ENVI expressions list
- **GetEnviValue** – is used to extract a value from single ENVI expression, accepts single argument “value” containing single ENVI expression, and saves extracted value to temp folder into envivalue.tmp text file

ENVI supports simple **arithmetic**, **relational** and **logical** operations listed below in order of execution priority:

- **Arithmetic:**
  - (*) – Multiplication
  - (/) – Division
  - (+) – Addition
  - (-) – Subtraction
- **Relational:**
  - (==) – Equals
  - (!=) – Not equals
  - (>) – Greater than
  - (>=) – Greater than or equals to

- (<) – Less than
- (<=) – Less than or equals to
- (!) – Logical negation

- **Logical**
  - (&) – Logical AND
  - (|) – Logical OR

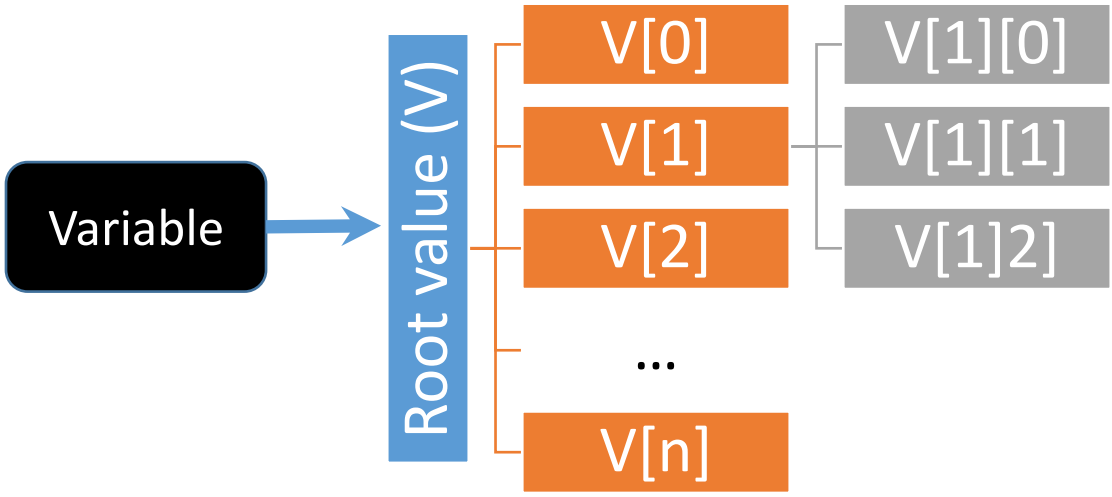
ENVI syntax is type-strong, i.e. a value of one type cannot be assigned to a variable of a different type without explicit conversion. The only exception is floating-point variables that can accept integer values. However, when an integer value is assigned to a floating-point variable, the value is converted to a floating-point data type.

All arithmetic and relational operations are applicable to **integer** and **floating-point** values. Some arithmetic and relational operations are applicable to **string** values (+, ==, !=). Logical operations are applicable to **Boolean** values only. Relational operation always return a **Boolean** value.

Array Trees

ENVI supports multi-dimensional vectors – variable capable of storing multiple values. The values are indexed by integer order index, similar to CSPro arrays. Structurally, these vectors are organized in a tree structure called **"array tree"**. Each variable has a root value, and each value contains a one-dimensional zero based vector of values of the same type with a variable length. The initial vector length is equal to zero. **Figure 1.3** describes the tree structure of value vectors.

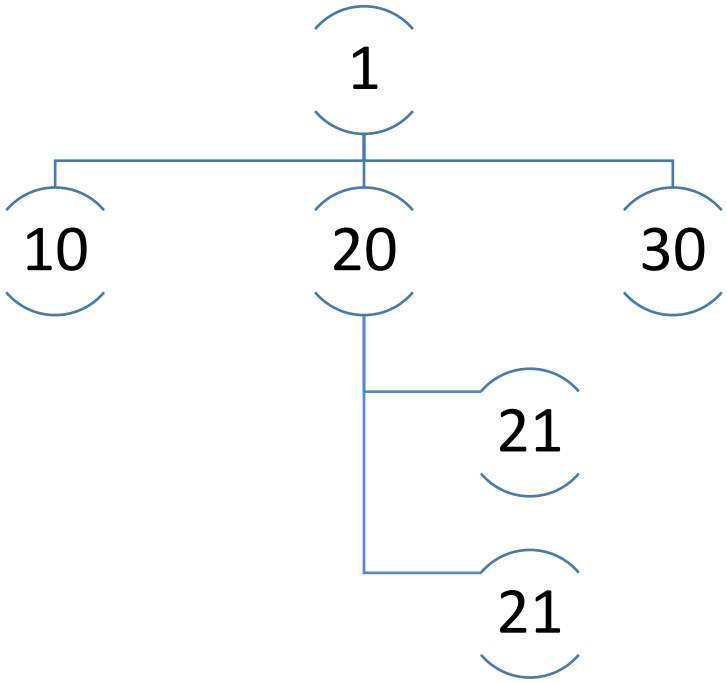
Figure 1.2. Value vector tree



Array trees have no limit on the number of branches, and, therefore, have no limit on a number of array dimensions. The following example demonstrates the construction and usage of the array tree:

```
Int A;
A = 1;
A += 10;
A += 20;
A += 30;
A[1] += 21;
A[1] + = 22;
```

The above example declares integer variable (A). Root value of (A) is set to (1). Operator (+=) is used to append additional values to vector branches of the value of (A). After the above code is executed, variable (A) will contain the following array tree:



In the above example expression **A[1][0]** returns integer value **21.** The declaration syntax can also be simplified to the following:

```
Int A = 1{10, 20{21, 22}, 30};
```

Composite Data Types

Additionally to basic data types, ENVI supports user defined composite data types called **"structures"**. Structures are declared with a unique name and can contain one or more variables of any type called **"fields"**. Because all ENVI variables are value types, circular structure declarations are not supported,

meaning a structure field cannot be of the same type as the structure itself, or any parent structures. Below is an example of an ENVI expression used for structure declaration:

```
struct CustomType {int I, string S};

CustomType CT;

CT.I = 10;

CT.S = "Hello World";
```

The above example declares structure with the name (**CustomType**) which contains two fields: integer field (I) and string field (S). After the structure is declared, we can now declare a variable (**CT**) of this new composite type and assign the values of its fields. Fields of the structure variable are accessed by using (.) operator. Structure fields can be of basic type as well as other composite type.

Menu

The menu visual component is designed to display a list of items to a user and to let the user choose from this list. Each menu item consists of a numeric integer ID and a specially formatted label text. The menu items can be organized into a tree list, where an item may open a list of sub items. In this case the item will display a **">"** symbol to the right of the label text, indicating that this particular item contains sub items. There is no restriction on the number of levels in the item tree.

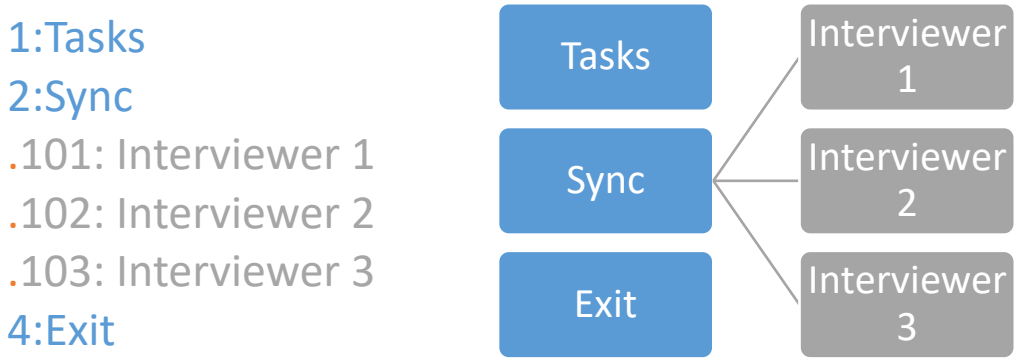
Figure 1.3.1. Menu window screenshot

Environment command **showMenu** is used to create and display a menu window. This command accepts a list of option parameters. Each option parameter represents a list item. Apart from item text, an option can supply an item ID as well as text color information. The option text is formatted using the following syntax:

[Level][ID:][textcolor=color;;;]Option text

- Level – level of item inside tree. For top level items it is empty. For levels below the top level is expressed in successive '.' symbol, one per each level. Items on the second level have '.', items on the third – '..', fourth '...' and so on. If the current item level is higher than the previous item level, the item is assumed to be a sub item of the previous item. **Figure 1.3.2**. describes the relationship between level item level indicator and position of item in a tree.

Figure 1.3.2. Relationship between an options list and items tree



- ID –custom numeric ID number the menu component will return to client application when the option is selected by user
- Textcolor –custom option text color. The color can be expressed as one of the system colors or as an RGB formatted color string ***(#RRGGBB).***

Note: See **Appendix II** for a complete list of system colors (p. 50).

When a user clicks on the menu option, the component returns a numeric option ID back to the client application. Additionally to user defined options, the menu contains a **Back** button at the bottom of the window. The showMenu command provides an option to encode a custom numeric ID number to return in case the user clicks this button.

Task List

The task list component is a set of visual tools and data handling routines designed to work with the task driven workflow model. The main idea behind the task driven workflow is to provide the user with a list of tasks and options to complete them. The task list component uses an underlying CSPro data file to store the tasks data, and, therefore, all task data manipulations can be performed from within a CSPro application. Static task metadata is stored in separate XML formatted files – task templates. Templates are used to categorize different types of tasks as well as define the task behavior and appearance.

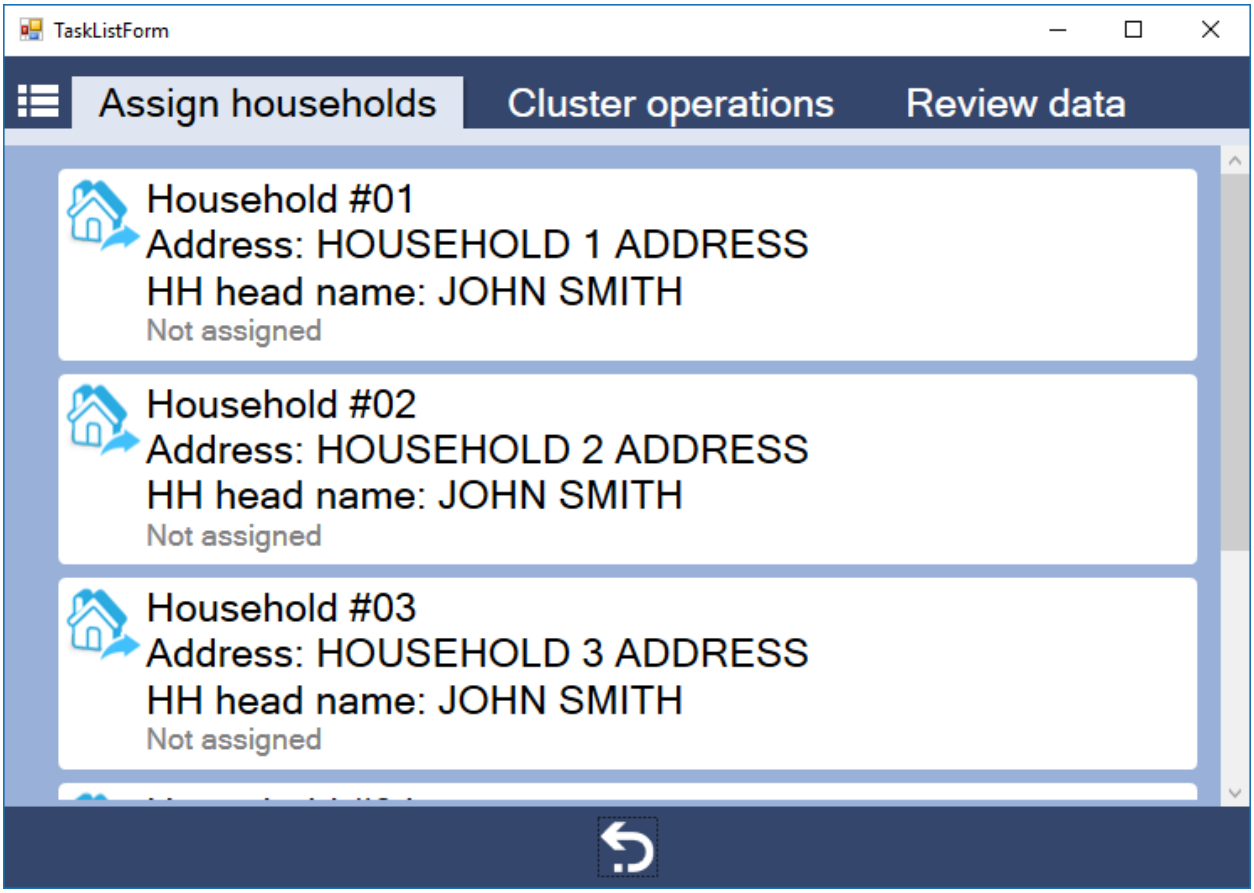
Tasks data file can store multiple tasks and is represented by a dictionary (TASKS.DCF). The list of items in the dictionary is presented in **Figure 1.4.1.**

Figure 1.4.1. List of items in the dictionary

Field	Description
TASK_ID	Unique identifier of a task within tasks data file. Usually 32-character long randomly generated Globally Unique Identifier (GUID).
TASK_TEMPLATENAME	Corresponding task template name
TASK_LABEL	Task label to be displayed to user
TASK_STATUS	Numeric value representing status of task
PARENT_ID	Optional field to define a parent of a task. Task list control displays tasks in a form of a tree. If the value of this filed is equal to the TASK_ID of another task, the visual control will nest the task under the parent task, independently of where in the data file the task is physically located
TASK_VARIABLES	Multiple group containing optional dynamic variables related to task
TASK_VARNAME	Dynamic task variable name
TASK_VARVAL	Dynamic task variable value
TASK_DUMMY	Dummy field at the end of the record, which prevents CSPro from shifting modified task records to the bottom for the file. Always needs to be set to a numeric value. For example, “0”.

The task list visual control is a window that is used to display task items and allow users to interact with tasks, as shown in **Figure 1.4.2.**

Figure 1.4.2. Screenshot of task list window



The task list window (or form) is used to display one or more task lists separated into tabbed panels. Each task list has a corresponding tasks data file where each case represents a single task. The appearance and behavior of task items inside a task list is dictated by the task template file. The task template file defines the following properties of a task list:

- Task list panels
- Task templates
- Task statuses
- Icons/fonts/colors
- Task options

Task options define a list of choices presented to the user when the task item is clicked on. Each option has an integer ID number defined in the template file. When the user clicks on an option, the task list window is closed and the control returns the ID of both the task and the option clicked, which can be used to route the execution of the application. If the task status does not have a defined options list, the returned option ID is equal to zero. Task template file can be created using the task template designer utility.

Task Variables

Each task item can store custom string values called task variables. In the task dictionary task variables are defined by a multiple group – TASK_VARIABLES. By default this multiple group has 16 occurrences, i.e. each task item can declare up to 16 task variables. Each task variable has a string name and string value. Task variables are used to store task specific information which can be accessed from within a CSPro application as well as can be used to direct task behavior.

For example, we can define a task for a household interview. The same interviewer may have several household assigned to them, and each assigned household will have a corresponding task in a task list. Although all these tasks are of the same type, some of the task parameters are unique to each task such as household number, household address, subsample information, etc. All of these parameters are attached to any particular task in a form of task variables. In our scenario we can store household number in an occurrence one of the TASK_VARIABLES group, where TASK_VARNAME(1) is equal to "hh_num" and TASK_VARVAL(1) is equal to "01". When dealing with any given task, we can read the value of the hh_num task variable inside our application and, therefore, get the household number from the task item.

Values of task variables can also be displayed in task labels and task status text as well as options and warning text of the task list. In order for the task variable value to be displayed in the task list, the task variable name must be included into the text enclosed in **"<%"** and **"%>"** characters. This is similar to how custom values are displayed in a QSF text of the CSPI CSPro entry applications.

For example, if the label text of the task is equal to **"*Interview household <%hh_num%>*"**, and the task variables group contains variable **"hh_num"** equal to **"01"**, then the resulting task label will be displayed in the task list as follows: **"*Interview household 01*"**.

Additionally, numeric task variables can be formatted as integer or floating-point value. For example, ***<%d:hh_num,3:000%>*** will result in a household number to be formatted as integer of three characters long with leading zeros.

Task Options

Task options are defined in a task template and are displayed in a sliding panel when the user interacts with the task in the task list by clicking or touching it. Task options have the following four properties:

1. **ID** – integer number identifying option within template. The number must be unique.
2. **Label** – text to be displayed for option. May include task variables.
3. **Warning text** – optional text to be displayed in yes/no warning message when user selects option. If warning text is empty, no warning message is displayed. Clicking on the **No** button in the warning message prevents from the task list returning to the menu application.
4. **Visibility flag** –Boolean variable defining visibility status of option. This flag can be used to dynamically hide an option from to be displayed to the user. The flag value is parsed as an ENVI logical or relational expression. It may also contain task variables. Task variables are evaluated before the flag is processed by the ENVI module.

Let us consider the following example: We have a task to interview an individual of either male or female gender. The visibility flag of an option is set to **"*ig==<%rsex%>*"**. Let us assume that the gender of the interviewer is stored in an ENVI variable **"*ig*"** and is equal to either **"*1*"**or **"*2*"**. Similarly, the task

variable **"*rsex*"** stores the gender of the respondent. Therefore, the visibility flag ENVI expression will be evaluated to either **"*true*"** or **"*false*"** based on the equality condition between the gender of the interviewer and the gender of the respondent. Thus, the option will only be visible to the interviewer, if their gender corresponds to the gender of the respondent.

Bluetooth Synchronization

Bluetooth synchronization component is used to transfer files or tasks between two devices. Synchronization can only take place if both devices have Bluetooth radios enabled. Otherwise synchronization fails.

Before files or tasks can be transferred between users, they have to be packaged and placed in a transfer cache file. At the time of sync component initialization, the application must supply a path to the cache file. If cache file does not exist, it is created automatically. The cache file serves as an intermediate repository for all data to be transferred by the sync component. When the application stores files or tasks to be transferred to another device, it must create a package to be stored in the cache. Apart from the data itself, the package contains the following identification information:

- **Content ID** – optional user defined string identifier of package content
- **Sender ID** –unique string identifier of sending user (user that is creating package)
- **Receiver ID** –unique identifier of receiving user

Once the package is created, it is stored in the cache file. (See **Figure 1.5.1.** for more detail.) The cache file defines several shelves, on which the package can be stored. At the time of package creation, the client application must supply a shelf name on which the package will be stored in the cache. Many of the shelves are not exposed to the client application, as they are used internally by the sync component. However, the following shelves are available for the client application for package storage and processing:

- **Outgoing** – shelf used to store all packages to be sent out to remote clients. After two devices connect and complete the handshake protocol, all packages from the outgoing folder are transferred over to their destination clients based on the package ID information. After the transfer is complete, the packages are removed from the outgoing folder of the sending client.
- **Received** – all incoming packages from remote clients are initially stored on this shelf. The sync component processes all packages on this shelf automatically. But the client application has functionality to invoke unpacking from this folder manually as well.

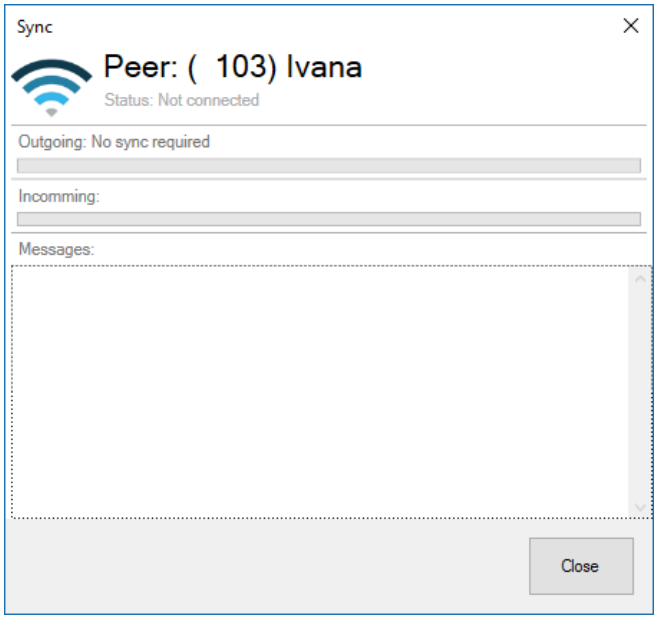
Figure 1.5.1. Common scenario for synchronizing data

Note: See **Appendix I** for a complete list of functions that work with the Bluetooth synchronization component (pp. 31-49).

Bluetooth Synchronization Window

When two users initiate the synchronization via the Bluetooth component, the system displays the synchronization window as shown in **Figure 1.5.2**.

Figure 1.5.2. Synchronization window



This window displays the connection status to the remote peer and transaction progress. When both peers initiate the transfer, the system attempts to establish a wireless connection between two devices. In order for this to happen, the client application must know the Bluetooth MAC address of the remote peer before connection can be established. Once the connection is established, the status of the connection will change to **"Connected"** and the transaction will proceed. In case if there are outgoing packages, they will be transferred to the remote client and outgoing progress will reflect the status of sending outgoing packages. If there are incoming data, the incoming progress bar will display the receive progress as well as display messages on successful receipt of data. After all incoming and outgoing packages are processed by both peers, the window will close automatically.

Types of Packages

The system is designed to natively synchronize tasks and files between users. In **Figure 1.5.1.** the procedures for generating packages differ depending on whether the file or task package is being

created. After the package is created and stored on the shelf in a cache, all further steps in the synchronization process are universal, and the sync component does not differentiate between the type of data in transit. The sections below describe the procedures for synchronizing tasks and files.

Synchronizing Tasks

A task is a record in CSPro data. Transferring a task via the Bluetooth sync component requires the following steps:

1. **Convert the task data into a predefined ENVI structure.** During this step, the client application must define an ENVI variable of type **"*task*"**. This composite type (structure) is declared by the sync component and contains all fields to describe a task (similar TASKS.DCF dictionary). The task structure is declared in the following format:

```
struct task {
        string id,
        string templatename,
        string label,
        int status,
        var variables {
                string name,
                string val
        }
};
```

   All fields of this structure mimic the tasks dictionary. Field **"*variables*"** is used as a tree array to store multiple task variables.
2. **Create a sync package using the data from the ENVI structure by calling the "*addTaskToSync*" environment command.** Here besides the task ENVI variable, the client application needs to supply the destination path for the task. The task will be merged into a CSPro data file defined in this path on the receiving end of the transaction. If the file doesn't exist, it will be created.
3. **Synchronize with remote client.** On the receiving end, the client application must supply the path to the tasks CSPro dictionary, which will be used to reconstruct task record before writing them to the specified destination path.

Synchronizing Files

Besides tasks, the sync component can transfer files via Bluetooth between two piers. The steps to package the files for transfer are as follows:

1. **Create an ENVI variable of composite type "fileslist" which represents a list of files to be transferred.** During this step the client application must define an envy variable of type "**fileslist**". This composite type (structure) is declared by the sync component and contains all fields to describe a task (similar TASKS.DCF dictionary). Task structure is declared in the following format:

```
struct fileslist {
        filerec files{
                string source,
                string dest
        }
};
```

The fileslist structure contains only one field of another composite structure type **"filerec"**. The filerec structure contains metadata required to transfer one file, and declares the source and destination paths of this file. Field files of type **"rilerec"** may contain an array tree of multiple files, and, therefore, allows the sync component to transfer multiple files in one transaction.

2. **Create a sync package using the data from the ENVI structure by calling the "addFilesToSync" environment command.** This command expects a name of the fileslist ENVI variable as input as well as other package ID information.
3. **Synchronize with remote client.** Once the file package is delivered to the destination pier, the files are automatically unpacked and saved to their respective destination paths.

Package Name Filters

The sync component is designed to deliver any package from the outgoing cache shelf to the currently connected remote pier. Although this feature is very powerful in terms of transaction management, it also requires the client application to actively manage which packages in the local cache are to be transferred or not transferred at the time of synchronization. This is accomplished by using **"package name filters"**.

Package name filter is a string that defines which currently available packages are to be transferred over based on the content of their names. A package name is also a string, consisting of multiple substrings delimited by double semicolon "***;;***". There are three substrings:

1. Package content ID
2. Receiver ID
3. Sender ID

Therefore, for example, if the package content ID is equal to **"content_id_123"**, the receiver ID is **"101"** and the sender ID is **"102"**, then the name of the package would be constructed as:

- ***content_id_123;;101;;102***

By default, during the synchronization procedure, all packages from the outgoing shelf are scheduled to be transferred to the connected pier. However, the sending pier may use the name filter to only submit certain packages to the transfer queue. Name filter string is constructed using substrings of double semicolon delimited values. Each value is an index/value pair delimited by a colon **":"**. The index potion represents a numeric zero based index of the value within the package name. For example index **"0"** will refer to the first element in the package name. In the above example this element is equal to **"content_id_123"**. Index **"2"** will refer to the third element, which is equal to **"102"**.

Consider the following example that demonstrates the name filtering in action. Suppose that we have three packages in the outgoing shelf:

- ***content_id_123;;101;;102***
- ***content_id_456;;105;;102***
- ***content_id_789;;105;;102***

When synchronizing with user 105, we only want to transfer packages related to that user, i.e. the receiver ID (second element in the package name) is equal to 105. Therefore when synchronizing with this user, we would also include the following package name filter:

- "***1:105***"

The sync component will analyze the packages names and will only queue for transfer the ones that pass the name filter. In this case, the filter only passes for packages two and three, as the second component of their name (zero-based component index "1") is equal to **"105"**. The first package does not pass the filter, and therefore is not transferred.

If we modify the filter string value to the following:

- "***0:content_id_456;;1:105***"

Then only the second package in our example will be queued for transfer, because only the name of the second package passes our modified name filter.

HTML Report Factory

The HTML report factory component is designed to simplify creation of HTML formatted documents and pages. The component utilizes a series of environment commands to construct an abstract layout of the document and generate HTML document structure with minimal knowledge of HTML architecture. The document is organized as a tree of elements. There are five different types of elements available:

1. Label –generic element designed to contain and display HTML text; implemented with HTML *** *** tag
2. List – displays HTML ordered ***<ul>*** and unordered ***<il>*** lists
3. Table – displays HTML ***<table>*** element
4. Progress bar – displays generic progress bar with value normalized to 100 by default
5. Hyperlink – displays HTML hyperlink ***<A>*** element

As mentioned above the document elements can be combined to form a tree, where one element can nest another element within itself. For example, a label element can nest a table within its content and a table cell can nest a list element. Each document element is created with a unique name. This name, if placed inside the content of another element with the **"%"** sign on both ends will inform the document builder component to nest one element within the content of another.

For example, if we create two tables with names **"table_1"** and **"table_2"**. We define table_1 to have three rows and four columns and table_2 to have two rows and two columns. When creating a cell in table_1 on row two and column three, if we include ***%table_2%*** into the content of that cell, table_2 will be nested within the cell (2,3) of table_1, as demonstrated in **Figure 1.6.**

Figure 1.6. Nesting of two tables

Table_1

		Table_2 <table><tr><td></td><td></td></tr><tr><td></td><td></td></tr></table>	

Document elements are created independently from one another. In the above example it does not matter if table_1 was created before table_2. All of the nesting is resolved when the report builder component is instructed to create (or render) the document. At that time a recursive dynamic process iterates through the entire set of created components and dynamically builds the HTML output.

Due to the recursive nature of the report building process, circular nesting is prohibited. The document tree must not have any loops of elements in it. Otherwise the document building process will fail. The root element of the document tree is a "page". Page content is populated using the **"addContent"** environment command. Once the document is constructed and all elements are created, the document can be written out into an HTML formatted file using the **"renderDocument"** command. The document content can be cleared using the **"clearDocument"** command.

Document Styling

Visual styling of elements in the document is achieved through HTML cascading style sheets (CSS). Every element in the document has a default HTML style, but can also use custom defined CSS. Style sheets files can be linked to the document using the **"addCssLink"** environment command. Multiple CSS files can be linked to the same document. Once the CSS file is linked to the document, custom defined CSS classes can be assigned to document elements. The **"clearCssLinks"** command removes all current CSS references from the document.

Tables

A table creation process consists of the followng three steps:

1. Create table element using the **"setTable"** environment command
2. Add table rows using the **"addTableRow"** environment command
3. Add table cell for the current row using the **"addTableCell"** environment command. Note that when a row is added to the table, the row becomes the "current row" and all added cell will be appended to that row. Therefore, table cells cannot be added independently from the table rows.

Progress bar

Progress bar is a non-native HTML element, and is constructed in the document using a combination of HTML tags. The two following commands are used to create and style the progress bar:

- **SetProgressBar** – creates progress bar element. The minimum value is zero and maximum is 100. Progress bar displayed value needs to be set anywhere between the minimum and maximum.
- **SetProressBarCss** – assigns CSS classes to each of progress bar sub-elements. There are three sub-element layers in the progress bar element: back frame, front frame, and text layer. Each one can be assigned a custom CSS class.

Data Grid

The data grid is an interactive visual component used for displaying data in a form of a table. Similar to the tasks list, this component displays a list of records originating from an underlying CSPro data file in a predetermined format. Unlike the tasks list, the data records are displayed in a table, instead of separate data items (tasks). Once the data grid is displayed, the user can select one of the data records and, similar to the task list, the component closes the data grid window and returns a data case ID for the selected record.

The data grid component also implements functionality for dynamic data filtering based on one of the data columns. Process container environment may store multiple data grids within one session.

As mentioned above, the underlying data displayed inside the grid comes from a data file. The table metadata is defined through environment data grid commands which are like all environment commands are accessible by the client application. The data grid metadata consist of the following:

- **Colum definitions**
  Each column inside the data grid has the following properties:
  - Name – name of the column. The name must be unique within the table.
  - Label – text to be displayed at column header
  - Visibility flag – Boolean flag which determines whether column is visible to user

  Column sequence in the table depends on the order of columns creation.
- **Filter items definitions**
  Optionally the data grid may filter out data rows based on string filters. These filters are defined using environment commands before the data is loaded into the data grid.
- **Filter column name**
  In case the data in the grid requires filtering, one column in the grid data has to be defined as a filter column. The name of this column has to be then assigned to the filter column name property of the grid before the grid data is loaded into the table.

Figure 1.7. Data grid with filtering

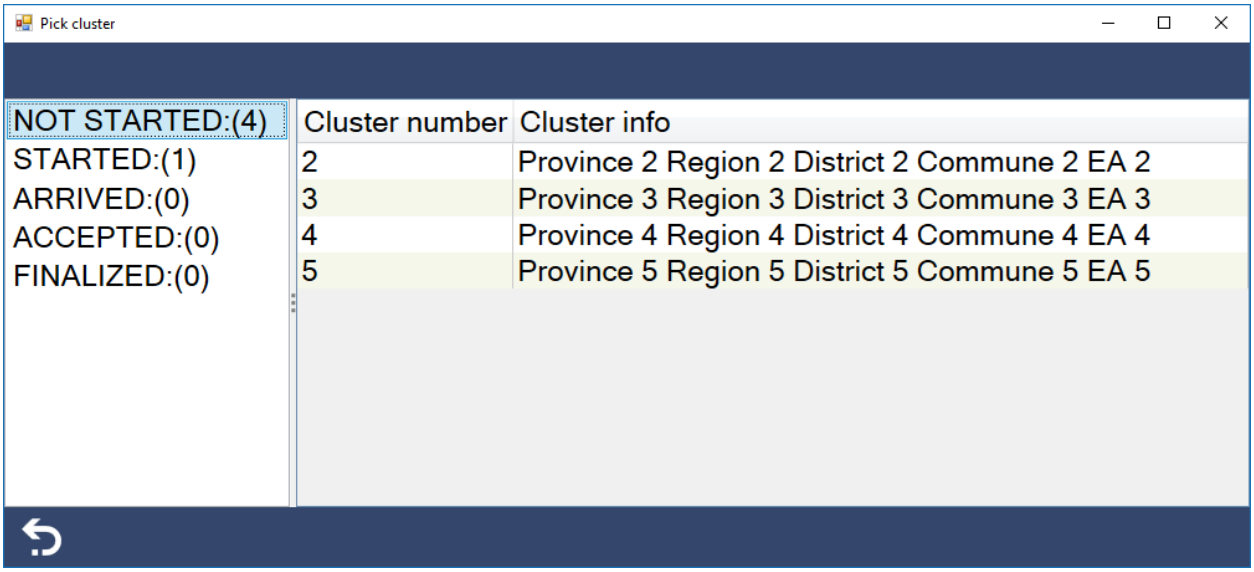


Figure 1.7. shows an example of the data grid window with filtering enabled. The table data is displayed in the right panel. Data filtering panel has five items defined and is displayed in a form of a list in the left panel. There are two visible columns in the data:

- Cluster number
- Cluster info

There is also a third hidden filter column which contains data corresponding to the filter items. Once the data is bound to the grid, the component dynamically compares the values of the filter column to the defined filters, and displays only those data records in which the data in the filter column corresponds to the selected filter item. In this particular example we see that there are four data records with filter value equal to "NOT STARTED" and one data record with filter value equal to "STARTED".

Note: Data filtering is optional. If no filtering is required and no filter items are declared, the left-side filter panel is dynamically hidden from the user.

After the table is displayed, the user can select one of the data rows by double clicking the row in the table, or selecting the row and clicking the **"→"** button in the bottom right corner of the window. Once the record is selected, the data grid window closes and the environment returns the data identifier of the selected case. This identifier is then passed over to the client application.

Grid Data Format

Data displayed in the grid comes from a CSPro data file formatted with static **"Grid.dcf"** dictionary. Just like with the task list, the client application is responsible for creating, maintaining and modifying the underlying grid data. The grid dictionary contains the following fields:

- ***GRID_ID*** –32-character long case identifier. Each case in a grid data file represents a row in the table. Therefore, each row in the grid must have a unique identifier***.***
- ***DGCELL*** –multiple group containing collection of cell in the row. By default, the maximum number of cells in a row is equal to 30. However, the maximum occurrence number in a group can be extended if needed. Each cell in the data consists two items as follows:
    - ***DGCOLUMN*** – grid column name. When the grid row is displayed by the grid component, the column names for each cell of this row are matched against the defined columns in the grid metadata.
    - ***DGVALUE*** – Cell value

Note: The order of the cell in the DGCELL group is not relevant to the order in which the data is displayed in a grid window.

Process Container Environment Debugging

Because the client application executes commands through an external call to the PCommand.exe module, there is no direct link between the debugging infrastructures of the client application and the process container environment. Therefore, it is not possible to debug environment related code during the compilation time of the client application. The environment debugging is only possible during runtime. For example, if the client application submits a syntactically incorrect ENVI expression to the environment, it is only possible to register the syntax error only during the execution of the command, which evaluates this expression. Because the ENVI expression is just a string inside the client application, the client application compiler will not detect the error at compilation.

By default, the environment attempts to execute a command, and if any error prevents it from being executed, the environment returns to the client application without executing the command. The debugging layer is disabled. Executing the **"SetDebugFile"** command enables the environment debugging. After debugging is enabled, every time a command execution raises an error, this error will be appended to a text file specified as debug output file.

Global modules

Starting with Version 6 CSPro provides functionality to plug in external logic modules to CSPro applications. Similar to main application logic, external modules are files with series of CSPro declarations that can be accessed from the main application. The main idea behind external modules is to reuse common static functionality without having to declare it in every application that uses it. External logic files can declare working variables as well as user defined functions. The MICS CAPI system makes use of the external logic modules to encode commonly used variables and functions.

Command Wrapper Module

As described above, the functionality of the process container environment is accessed through environment commands. These commands are URI formatted strings that are passed as command line arguments to the PCCommand.exe process. To simplify the command call procedures a common external logic module is included in the system, which serves as a CSPro wrapper to the execution of environment commands. Instead of constructing raw commands, this module provides a collection of CSPro user defined functions that can be invoked from any CSPro application. These functions are described in **Appendix I** of this document (pp. 31-49). The file name of the module is **"CommandWrappers.apc"**.

There are the following two global variables declared in this module that the client application must assign before the module logic can be invoked:

- **utilsDir** – path to directory with process container environment executable
- **tempDir** – path to temporary directory used by process container environment

Global Variables Module

The second module distributed with the system is called **GlobalVar.apc**. This module contains global declarations used by multiple applications throughout the system. The following essential global variables are declared in this module:

- **workDrive** – path to logical drive the system is installed on
- **projectDir** – CAPI project root directory
- **projectName** – CAPI project name
- **csproDir** – path to directory where CSPro is installed. By default is assigned using the **"pathname(CSPro)"** function.
- **projectStartYear** – data collection cycle start year
- **projectEndYear** – data collection cycle end year

All of these variables are assigned within a user defined function **SetGlobalVariables()**. The implementation of this function is project-specific and needs to be adjusted for every survey. The SetGlobalVariables() function must be called from every application using the module. Other commonly used functions declared in the global variables module are described in **Appendix I** (pp. 31-49).

CAPI Menu Applications

As mentioned in the beginning of this document, the MICS digital data collection system consists of a series of CSPro entry and batch edit applications. These applications are responsible for data management on all levels of the survey data collection operation. There are three distinct sub-systems:

- **Interviewer system** – deployed on the interviewer mobile device allowing for collection, management, and transfer of data by individual interviewers
- **Supervisor system** – deployed on the team supervisor mobile device allowing for team task management as well as accumulation of data collected by interviewers within the team and reporting on the progress and quality of collected data
- **Central office system** – deployed on a stationary computer at the survey central office location; serves as a data management platform for accepting and processing data coming from team supervisors. This system also provides comprehensive data collection progress and data quality reporting.

Any given survey presumes a set of questions that the interviewer is expected to ask the respondent and record the answers. This set of questions is typically organized in logically structured questionnaires. In CAPI surveys the role of the data collection software is to implement the logic of the questionnaires and provide the interviewer with an interface to ask the questions and record the answers with maximum precision. In the MICS data collection system, the role of the data collection software is played by specially developed CSPro entry applications. They mimic the structure and logic MICS paper questionnaires and provide comprehensive human error mitigation where possible. These applications are designed to record the interaction between one interviewer and one respondent at a time. However, the typical survey envelops thousands of respondents as well as a hundred or more interviewers. That is why besides data collection software, the MICS digital data collection system deploys a set of data management applications – menus.

If data entry applications govern the interaction between interviewers and respondents, then menu applications govern everything else in the data collection process that happens before and after the interviewer has conducted every interview. Because the system is distributed, i.e. multiple parts of the system run on multiple devices concurrently, menu applications are designed to manage this concurrent process for the entire data collection operation. Each subsystem interacts with its user providing a visual list of tasks and actions that the user is expected to perform.

Interviewer Menu

The interviewer menu application is a CSPro data entry application which is used as a main logical hub for data management on the interviewer level. This application serves as a client application to the process container environment and relies heavily of its components. The main application file is called **"InterviewerMenu.ent"** and it resides in the Applications\Interviewer subfolder of the project.

The main logic file of the interviewer menu is **"InterviewerMenu.ent.apc"**. This file consists of all functions used in the interviewer menu system. The reader of this document is expected to have the knowledge of CSPro and ability to read and understand CSPro code. This section describes some of the less intuitive algorithms and functions used in the interviewer menu application.

It is worth noting that all menu applications, including the interviewer menu, use the following common modules:

- **CommandWrappers.apc** – contains functions to simplify interaction between CSPro applications and PC environment. *(See **Appendix I** (pp.XX-XX).)*
- **GlobalVars.apc** – contains global common variables used in the system as well as some general functions. *(See **Appendix II** (p. XX).)*

The logic in PreProc of the application is responsible for setting global application parameters such as project folders. Instructions also include creation of team members array, task list, and synchronization cache.

The system automatically detects the user based on the device Bluetooth MAC address, and, therefore, it does not expect manual entry of interviewer credentials. The only piece of information entered manually is the cluster number; therefore, the rest of the menu processing is located in PostProc of the MCLUST field.

The logic runs a loop of the main menu, tasks list, and Bluetooth sync until the user exist the system. Every time the interviewer selects a task from the list, the application loads the task data into the TASKS dictionary and returns the option ID. Using this information the **"TaskRouter()"** function directs the further execution of the application. The following list describes some of the other functions declared in the interviewer menu application:

- ***SetTaskVar() / GetTaskVar()*** - sets and retrieves value of task variable for the currently loaded task
- **BackupData()** – copies the collected data files as well as tasks to all backup drives
- **FillTeam()** – constructs an array of interviewer team member with all related information
- **ShowTasks()** – displays task list window and in case user selects task/action, loads task data into TASKS dictionary and returns selected option ID
- **UpdateTasks()** – runs UpdateTasks.bch batch application, which updates task list based on currently available data
- **PackDataForSup()** – creates outgoing synchronization package containing current data and saves to sync cache file. Note that before creating new package, will delete the old package with the same content ID. This way only the latest data is included in the transfer.
- **GenHhIntPff()** – starts household entry application
- **DeleteHousehold()** – deletes all household data, including all individual questionnaires. Also deletes all relevant tasks from the task list.
- **GetSplitHhNum()** - generates new household number for split household. Returns 0 if cannot generate new HH number.
- **SplitHousehold()** – generates new task in task list for split household
- **GenIndIntPff()** – starts individual entry application
- **Delegate()** – delegates individual interview to another interviewer
- **DeleteIndividual()** – deletes individual case, but leaves individual interview task present in list
- **BtSync()** – initiates Bluetooth synchronization process with another team member
- **SetMenuColors()** – dynamically sets colors for main menu options based on synchronization status

UpdateTasks.bch Application

Each interviewer maintains a list of assigned tasks. By completing these tasks the system keeps track of the task statuses as well as generates new tasks based on the requirement of the survey. In order to do so, the system needs to have access to the collected data and, therefore, to all of the CSPro dictionaries related to all questionnaires. To reduce the size of the menu application and the number of referenced external dictionaries, a separate batch application is used to analyze the data collected by the interviewer and update the task list data file. This application is called **"UpdateTasks.bch"** and the menu calls this application every time the user performs an action which may affect the questionnaire data.

Supervisor Menu

Similar to the interviewer menu, the supervisor menu application is a CSPro data entry application that is used as a main logical hub for data management on the supervisor level. This application serves as a client application to the process container environment and relies heavily of its components. The main application file is "**SupervisorMenu.ent",** and it resides in the Applications\Supervisor subfolder of the project.

Structurally the supervisor menu application logic is similar to the interviewer menu; however, the tasks performed by the supervisor are different from the tasks performed by an interviewer, therefore the functions comprising the supervisor menu application are also different. The following list describes selected functions declared in the supervisor menu application logic file:

- **genTasks()** – generates task list data file for cluster. This function is called when the supervisor opens a new cluster for data collection. The function will generate household assignment tasks as well as general tasks related to cluster processing on the supervisor level.
- **onHhAssign()** – handles event of household assignment by supervisor to interviewer. This function will update the household assignment task as well as create a household collection task for the interviewer and add it to the sync cache file.
- **concatIntervData()** – concatenates data from all interviewers per cluster. The input is in the work folder, the output is in the receive folder.
- **onClustClose()** – handles cluster closing procedure. This function will call the CKID application and in case the data collection is complete for the cluster and no structural errors are detected, will change the task status to "Closed" for the cluster.
- **reviewHhInt()** – starts data entry application for existing household collected by one of interviewers. This function will create a temporary copy of the household data; therefore, any changes to the data will not be reflected in the actual data coming from the interviewer.
- **reviewIndInt()** - starts data entry application for existing individual collected by one of interviewers. This function will create a temporary copy of the individual data; therefore, any changes to the data will not be reflected in the actual data coming from the interviewer.
- **procNoteLine()** – function used to convert raw note record into HTML table row. This function is called from within the **reviewNotes()** function.
- **reviewNotes()** – compiles all notes taken by the interviewer into HTML report for household, including all individuals collected in this household
- **writeVerifTabl()** – creates HTML table for household schedule of one household during verification by supervisor

- **compareVerifHhSchedule()** – generates HTML report comparing household schedules of data collected by interviewer and verifying data collected by supervisor
- **taskRouter()** - main function that makes decisions on routing code execution for currently loaded task
- **getReviewedHouseholds() -** creates a semi-colon separated string with all household cases found in the "R" file (supervisor verification file)
- **populateReviewTasks()** – populates task list data file displayed on review data panel
- **syncCentral()** – initiates remote data synchronization procedure with central office server
- **syncCentralRequired()** – checks if synchronization with central office is required
- **setLastDateMarker()** - setting date marker after successful remote synchronization with central office
- **doUpdate()** – checks update folder content. If there are files to update, updates the current system and creates update packages for all interviewers on the team.

There are two additional external batch applications used by the supervisor menu: **CKID.bch** and **GenReviewTasks.bch**. Similar to how the **UpdateTasks.bch** application implements some of the functionality of the interviewer system to reduce the load on the main menu application, these two applications accomplish tasks that would make the main supervisor application too bulky and hard to maintain if included as part of the main menu logic.

CKID.bch

This batch application is used to analyze the structure of the data collected in a cluster and provide a detailed progress report on data collection operation. It will also flag all structural errors in the data and report it to the supervisor. The menu application will not allow the supervisor to close data collection in a cluster unless all errors are resolved and all identified questionnaires are collected. This application uses the report factory environment component to produce data collection progress reports.

GenReviewTasks.bch

This batch application is used to generate a task list data file for reviewing questionnaires collected by interviewers. The logic is similar to the **UpdateTasks.bch** application; however, the difference is that **GenReviewTasks.bch** does not account the sampled unvisited households.

Central Office Menu

The central office menu application is a CSPro data entry application which is used to receive and process data at the central office level. The main difference from the interviewer and supervisor menu is that typically this application is deployed only on one computer instead of multiple mobile devices. This application serves as a client application to the process container environment and relies heavily of its components. The main application file is **CentralMenu.ent** and it resides in the Applications\Central subfolder of the project.

The central office system is designed to accomplish two main tasks during the CAPI data collection process. One is to receive the data coming from the field supervisors and track progress of data collection as well as data quality. Second task is related to finalization of data that includes secondary editing, review, and data exporting procedures.

Structurally, the central office menu application logic is similar to the interviewer and supervisor menus. The following list describes selected functions declared in the central office menu application logic file:

- **genNewTasks()** – generates new task data file of clusters that have been closed in the field, but not yet accepted at central office
- **addProcTasks()** – generates additional tasks for cluster after it has been accepted at central office
- **getClusterStatus()** - returns clusters status based on data files in folders. 0 - not started, 1 - started, 2 - arrived, 3 - accepted, 4 – finalized.
- **genGridDataFile()** - generates data file for data grid based on clusters file
- **pickCluster()** – populates data grid with clusters and displays data grid
- **copyCluster()** – copies all cluster-related files from one folder to another
- **addEvent()** – adds event record to events file
- **progressReport()** – generates data collection progress report
- **exportData()** – exports data to SPSS

StatusReport.bch

StatusReport.bch is a batch application designed to produce a report on the status of the data collection operation. The report consists of overall survey data collection progress in percentages, a table of data collection status by cluster, and a table of data collection progress by interviewer team.

Events File

Events file is a CSPro data file formatted by the **Events.dcf** dictionary. This file is used to store records of events by cluster. The identifier field in the dictionary is called EVID and represents an event context. In the case of standard MICS the context is cluster, i.e. each case in the events file represents a list of events for one cluster. Record EVENT_REC is a multiple record with a maximum number of occurrences equal to 999. Each occurrence represents an event. Each event record consists of event type, date, time and optional alpha parameter. The central office menu application populates the events data file during execution of central office processing steps. Then the event file is used to reconstruct the events in time for each event context (cluster). The event file is a replacement for the old concept of control files. The advantage of the event file is that it adds a time dimension to the control data. Using the events file, it is very easy to provide reporting on certain aspects of the data processing in a given time frame.

Appendix I: List of Commands and Functions

SYSTEM COMMANDS			
Command	**Description**	**Parameter [] - optional ^ multiple**	**Return Value**
Start	Starts Windows executable process (.EXE) and attempts to dock process main window inside host window	* proc - path to executable file * [arg]^ - command line arguments * [cls] - main window class name * [title] - main window title portion * [caption] - custom window caption	
Exec	Starts Windows executable process (.EXE) without docking process main window inside host window	* proc - path to executable file * [arg] - command line arguments string * [window] - main window state: normal, max, min, hidden	
Messagebox	Displays standard Windows message box dialog window	* [caption] - message window caption text * [text] - message text * [buttons] - message box buttons combination: AbortRetryIgnore, OK, OKCancel, RetryCancel, YesNo, YesNoCancel * [icon] - message box icon: Asterisk, Error, Exclamation, Hand, Information, None, Question, Stop, Warning	
Sleep	Pauses execution of main command thread by specified number of milliseconds	* delay - milliseconds number	
Settempfolder	Sets global system-wide temporary folder	* path - physical folder path	
Shownotifyballoon	Displays standard windows notification balloon message	* [caption] - message caption * [text] - message text * [icon] - message icon: Application, Asterisk, Error, Exclamation, Hand, Information, Question, Shield, Warning, Winlogo	

SYSTEM COMMANDS			
Command	**Description**	**Parameter [] - optional ^ multiple**	**Return Value**
Runcommandslist	Sequentially executes list of process container commands stored in text file. Expects one command per line.	* file - path to text file with list of commands to run	
Setdebugfile	Sets debug output file to which system will write all debug messages	[debugFile] - path to debug output file. (If parameter not included, the system disables debugging for the current session.)	
Writedebugmessage	Writes test message to debug file. Only works if debug file is set	* message - message to write out to debug file	
getsystemdrives	Scans current system and returns list of mapped drives	* [removableonly] - if set to more than zero, returns only removable drives	Returns a semicolon delimited string of all discovered system drives. The output is saved to the "systemdrives.tmp" file in the environment temporary folder.
exitenvironment	Terminates backgroud 'PContainer.exe' process		
proccount	Returns count of running processes with given executable name	* exename - process executable file name	The output is saved to the "proccount.tmp" file in the environment temporary folder

SYSTEM FUNCTION			
Function	**Description**	**Parameter [] - optional ^ multiple**	**Return Value**
runCommand (string command)	Executes command with raw environment syntax. (If buffered execution is enabled, adds command	* command - command syntax to execute	

SYSTEM			
FUNCTION			
Function	Description	Parameter [] - optional ^ multiple	Return Value
	to buffer file.)		
setTempFolder (string tempFolder)	Setting temp folder for container app	* tempFolder - path to temp folder	
loadPreviousValues()	Reads state.dat file in temp folder and populates array of previously saved value		
savePreviousValues()	Writes out array of values to state.dat file		
enableDebugMode (string debugOutput)	Enables debug mode where all command errors are appended to debug output file	* debugOutput - path to debug output file	
disableDebugMode()	Disables debug mode		
writeDebugMessage (string message)	Writes test message to debug file. Only works if debug mode is enabled.	* message - message to write out to debug file	
openBuffer (string bufferFile)	Opens buffer for buffered command execution	* bufferFile - path to file that will store list of commands for buffered execution	
closeBuffer()	Closes command buffer and executes all recorded commands		
string getSystemDrives (removableOnly)	Scans current system and returns ';' delimited list of mapped system drives	* removableOnly - if not equal to zero, function returns only removable drives	
execProc (string target, string arg, string windowMode)	Executes and external process (similar to execSystem() function in CSPro)	* target - path to target executable * arg - argument string * windowMode - normal, maximized, minimized, hidden (normal by default)	
exitEnvironment()	Terminates running process container environment instance (PContainer.exe)		
procCount (string procName)	Returns number of running Windows processes with given name	* procName - process executable file name	Returns number of currently running processes

ENVI COMMANDS			
Command	**Description**	**Parameter [] - optional ^ multiple**	**Return Value**
execenvi	Executes list of ENVI commands	* commands - ENVI formatted syntax string * [caption] - custom window caption	
getenvivalue	Parses ENVI expression and returns value. The output value is saved to temp folder with filename envivalue.tmp and contains a string representation of the value. If parse fails, writes out '*'.	* value - ENVI expression	

ENVI FUNCTION			
Function	**Description**	**Parameter [] - optional ^ multiple**	**Return Value**
execEnvi (string commands)	Executes ENVI syntax string	* commands - semicolon delimited ENVI commands string	
string readEnviValue (string expr)	Reads value from ENVI expression	* expr - ENVI expression to evaluate	Returns string with ENVI value. Returns empty string if unable to evaluate expression.

MENU COMMANDS			
Command	**Description**	**Parameter [] - optional ^ multiple**	**Return Value**
showmenu	Displays menu dialog window	* [title] - menu dialog window title text * [fontSize] - menu items font size * option^ - list of menu formatted items * [backButton] - optional return value for 'back button'	Returns selected item ID index. The output is saved to the "menuform.tmp" file in the environment temporary folder.

MENU			
COMMANDS			
Command	Description	Parameter [] - optional ^ multiple	Return Value
setmenucontraints	Sets size constrains on menu window. Window size is expressed in percentages relative to screen size.	* [minx] - minimum width of window * [maxx] - maximum width of window * [miny] - minimum height of window * [maxy] - maximum height of window	

MENU			
FUNCTION			
Function	Description	Parameter [] - optional ^ multiple	Return Value
setMenuConstraints (minX, minY, maxX, maxY)	Setting % constraints on menu relative to screen resolution. By default menu will adapt to content size.	* minX - minimum horizontal menu size * minY - minimum vertical menu size * maxX - maximum horizontal menu size * maxY - maximum vertical menu size	
showMenu (string menuTitle)	Showing menu from array of options (1 based)	* menuTitle - menu window title	
clearMenu()	Resets all elements of 'menuOptions' array to empty string values		
addMenuOption (string optionText)	Appends new menu option to 'menuOptions' array	* optionText - appended menu option text value	
addMenuSeparator()	Appends separator after last current menu option		

TASKS			
COMMANDS			
Command	Description	Parameter [] - optional ^ multiple	Return Value
createtasklist	Creates new task list	* name - unique name of task	

TASKS			
COMMANDS			
Command	**Description**	**Parameter [] - optional ^ multiple**	**Return Value**
	object in current environment instance. Multiple task list object may be maintained within one environment instance.	list object. * template - path to task template file	
loadtasks	Loads tasks data file into task list object	* tasklist - previously created task list object name * panel - zero based index of task panel, which will be populated with loaded tasks * dcf - path CSPro dictionary file defining tasks data format * data - path to CSPro data file containing task records to be loaded into task list	
showtasklist	Displays task list window	* tasklist - previously created task list object name	Returns a semicolon delimited string with three task identification items: * Task ID value in the loaded tasks data file * Selected option ID. (If no options defined returns (-1).) * Zero based panel index The output is saved to the "tasklist.tmp" file in the environment temporary folder.
setoption	Dynamically sets option for existing task template. Only valid for currently running environment instance. Does not	* tasklist - previously created task list object name * id - numeric option ID * label - label text * [warning] - optional warning	

TASKS			
COMMANDS			
Command	Description	Parameter [] - optional ^ multiple	Return Value
	update template file.	text * [visible] - visibility flag	
setoptionstree	Dynamically defines options tree in task status declaration for existing task template. Only valid for currently running environment instance. Does not update the template file.	* tasklist - previously created task list object name * template - task template name * status - numeric ID of status within task template * options - options tree index string	
cleanuptaskfile	Physically removes deleted task records from task data file and attempts to restore original task order.	* file - path to task data file * idlength - task identifier length * [skipdeletedcases] - if greater than zero, deleted task records are not written to output	

TASKS			
FUNCTION			
Function	Description	Parameter [] - optional ^ multiple	Return Value
createTaskList (string name, string templateFile, string output)	Creates new tasklist component object	* name - task list object name. Multiple task lists are supported by one environment instance, but names must be unique. * templateFile - path to XML task template file * output - path to file with task list output values	
loadTasks (string taskListName, panelIndex, string tasksDict, string tasksData)	Loads tasks data file into task list component object	* taskListName - name of tasklist to load data into * panelIndex – zero-based index of panel in task list window * tasksDict - path to Cspro dictionary that describes format of tasks data file * tasksData - path to tasks data file	
showTaskList (string taskListName)	Displays tasklist to users. Saves task and options	* taskListName - tasklist component object name	

TASKS			
FUNCTION			
Function	Description	Parameter [] - optional ^ multiple	Return Value
	ID after users select task and/or option.		
setOption (string taskList, optionId, string optionLabel, string optionWarning)	Dynamically sets option in task template	* taskList - task list name * optionId - option integer ID * optionLabel - option label text * optionWarning - option warning text. If empty string, no warning is displayed.	
setOptionsTree (string taskList, string templateName, statusId, string optionsStr)	Dynamically sets option tree for task status	* taskList - task list name * templateName - task template name * statusId - numeric status ID * optionsStr - string encoding new option tree	
cleanupTaskFile (string taskFile, idLength)	Helper function to remove deleted entries from tasks data file and sort it according to previously deleted order	* taskFile - path to CSPro task data file * idLength - length of ID variable in tasks dictionary (32 by default)	
cleanupTaskFile2 (string taskFile, idLength, skipDelCases)	Helper function to remove deleted entries from tasks data file and sort it according to previously deleted order with additional flag to keep or skip deleted cases in data	* taskFile - path to CSPro task data file * idLength - length of ID variable in tasks dictionary (32 by default) * skipDleCases - if not equal to zero, function will not write deleted cases into output	

SYNC			
COMMANDS			
Command	Description	Parameter [] - optional ^ multiple	Return Value
setcache	Sets data file for sync object. If file already exists, connects to existing file, otherwise creates it.	* filename - path to cache data file	
addtasktosync	Adds task package to warehouse database	* contentId - unique ID for package content	

SYNC COMMANDS			
Command	**Description**	**Parameter [] - optional ^ multiple**	**Return Value**
		* receiverId - receiving peer identifier * senderId - sending peer identifier * destFile - path to destination data file where task will be merged into * taskVar - environment variable with task data * [shelf] - shelf in cache where package is created. ('Outgoing' by default.)	
getbtaddress	Retrieves Bluetooth address of current device and saves it into text file	* outputfile - path to where Bluetooth address is saved	
removefromsync	Removes all packages with same content ID, receiver ID, and sender ID	* contentId - unique ID for package content * receiverId - receiving peer identifier * senderId - sending peer identifier * [shelf] - shelf in cache where package is located. ('Outgoing' by default.)	
getpackagescount	Returns count of packages in cache warehouse on specific shelf with optional package name filter	* shelf - warehouse shelf name * [filter] - formatted package name filter string	The output is saved to the "PackageCount.tmp" file in the environment temporary folder.
procincomingpackages	Processes packages in 'incoming' shelf	* filter - formatted package name filter string * taskdict - path to CSPro tasks dictionary	
addfilestosync	Adds file package to cache warehouse	* contentId - unique ID for package content * receiverId - receiving peer identifier * senderId - sending peer identifyer * encrKey - encryption key	

SYNC			
COMMANDS			
Command	**Description**	**Parameter [] - optional ^ multiple**	**Return Value**
		* filesVar - environment variable with files meta data * [shelf] - shelf in cache where package is created. ('Outgoing' by default.)	
createfileslistvar	Creates ENVI variable of type 'filerec', which holds file records for every file in specified source directory with specified mask	* filesvar - output ENVI variable name (if variable exists, will redeclare it) * sourcedir - source files directory * destDir - destination directory * [mask]^ - file filter mask * deep - if true, includes subdirectories (true by default)	
startsync	Starts synchronization with remote Bluetooth peer	* localaddress - local peer Bluetooth address * localpeerid - local peer ID * localpeername - local peer name * remoteaddress - remote peer Bluetooth address * remotepeerid - remote peer ID * remotepeername - remote peer name * tasksdict - path to dictionary that describes tasks * [filter] - filter string to filter out outgoing packages	

SYNC			
FUNCTION			
Function	**Description**	**Parameter [] - optional ^ multiple**	**Return Value**
string GetBtAddress()	Function to retrieve Bluetooth MAC address of current device		Returns string with current device Bluetooth MAC address

SYNC			
FUNCTION			
Function	Description	Parameter [] - optional ^ multiple	Return Value
setCache (string cacheFile)	Sets cache file for sync component	* cacheFile - path to cache file. If file doesn't exist, it will be created.	
addTaskToSync (string taskVar, string contentId, string receiverId, string senderId, string destFile, string shelf)	Adds task package to sync cache	* taskVar - ENVI structure with data for task to be added to cache * contentId - package content ID * receiverId - package receiver ID * senderId - package sender ID * destFile - path destination tasks file on receiver device * shelf - shelf to put newly created package in cache	
addFilesToSync (string filesVar, string contentId, string receiverId, string senderId, string encrKey, string shelf)	Adds files package to sync cache	* filesVar - ENVI structure with metadata for list of files to be included into package * contentId - package content ID * receiverId - package receiver ID * senderId - package sender ID * encrKey - encryption key used to encrypt files content * shelf - shelf to put newly created package in cache	
removeFromSync (string contentId, string receiverId, string senderId, string shelf)	Removes all outgoing packages with same content, receiver and sender IDs	* contentId - package content ID * receiverId - package receiver ID * senderId - package sender ID * shelf - shelf containing package	
startSync (string localAddress, string localPeerId, string	Starts Bluetooth synchronization process	* localAddress - local system Bluetooth address * localPeerId - local peer	

SYNC			
FUNCTION			
Function	Description	Parameter [] - optional ^ multiple	Return Value
localPeerName, string remoteAddress, string remotePeerId, string remotePeerName, string tasksDict, string filterStr)		identifier * localPeerName - local peer name (for display only) * remoteAddress - Bluetooth address of remote peer * remotePeerId - remote peer identifier * remotePeerName - remote peer name (for display only) * tasksDict - path to tasks dictionary file * filterStr - optional filter string to filter packages by content ID	
getOutgoingPackageCount (string filterStr)	Returns number of packages in outgoing shelf	* filterStr - filter string to filter packages by content ID	
procIncoming (string filterStr, string tasksDict)	Processes packages in incoming shelf. This usually happens after syncing automatically. But this function lets application process packages in the incoming shelf without synchronization.	* filterStr - filter string to filter packages by content ID * tasksDict - path to tasks dictionary file	
createFilesListVar (string varName, string sourceDir, string destDir, string mask)	Creates ENVI variable of type 'filerec', which holds file records for every file in specified source directory with specified mask	* varName - output ENVI variable name * sourceDir - source directory * destDir - destination directory * mask - file filter mask	

DOCUMENT BUILDER COMMANDS			
Command	**Description**	**Parameter [] - optional ^ multiple**	**Return Value**
cleardocument	Clears all nodes from documents and removes all content from pages, but does not reset title and CSS links		
setdocumenttitle	Sets document title	* title - document title text	
addcsslink	Adds CSS reference to document header	* css - path to css file (could be relative to document)	
clearcsslinks	Clears all CSS links in document		
addcontent	Appends HTML content to current page	* content - HTML content to be appended to document page	
clearcontent	Clears page content, but does not delete any elements from elements collection		
setlabel	Sets label element in element collection	* name - label element name. Must be unique within document * [content] - HTML content of label * [cssclass] - cssclass for label	
getlabelcontent	Returns label content string	* labelName - label element name	The output is saved to the "labelContent.tmp" file in the environment temporary folder.
setprogressbar	Sets progress bar in elements collection	* name - progress bar element name * min - minimum value of progress bar (0 by default) * max - maximum value of progress bar (100 by default) * value - current value of progress bar (0 by default)	
setprogressbarcss	Sets CSS classes for progress bar	* name - progress bar element name * css - class for overall progress bar element * cssBack - class for progress	

DOCUMENT BUILDER			
COMMANDS			
Command	**Description**	**Parameter [] - optional ^ multiple**	**Return Value**
		bar back element * cssFront - class for progress bar front element * cssText - class for progress bar text element	
settable	Sets table element in elements collection	* name - table element name. Must be unique within document * [cssclass] - cssclass for table	
addtablerow	Adds row to table element	* tablename - table element name * [cssclass] - CSS class for row * [th]^ - header cell text with default table CSS * [td]* - table cell text with default table CSS	
addtablecell	Adds cell to current row	* [content] - cell HTML content * [cssclass] - CSS class for cell * [isheader] - if true, writes out <th> tag instead of <td> * [colspan] - column span count * [rowspan] - row span count	
setlist	Sets list element in elements collection	* name - list element name * [cssclass] - cssclass for list * [isordered] - if true, creates ordered list	
addlistitem	Adds list item to list	* listname - list element name * [content] - list item HTML content * [cssclass] - list item CSS class	
renderdocument	Generates HTML file based on currently built document	* filename - path to HTML file output	

DOCUMENT BUILDER			
FUNCTION			
Function	**Description**	**Parameter [] - optional ^ multiple**	**Return Value**
clearDocument()	Clears document content		

DOCUMENT BUILDER			
FUNCTION			
Function	Description	Parameter [] - optional ^ multiple	Return Value
clearCssLinks()	Clears CSS links in document		
string htmlHeader (order, string content)	Creates HTML header tag	* order - header tag order value. Example: If order = 3, the tag is .. * content - HTML header content	
setDocumentTitle (string title1)	Sets document title to be displayed in title bar of browser window	* title1 - text to be displayed in title	
addCssLink(string css)	Adds CSS link to document	* css - URL to CSS file	
setLabel(string name, string content)	Sets label element in document. If label with name already present, clears its content	* name - label element name * content - HTML label content	
setLabelCss (string name, string content, string cssClass)	Sets label element with CSS class attribute	* name - label element name * content - HTML label content * cssClass - CSS class to be used with label element	
setTableCss (string name, string cssClass)	Sets table element with CSS class attribute	* name - table element name * cssClass - CSS class to be used with table element	
addTableRow (string tableName)	Adds row to table. The created row becomes the 'current row'.	* tableName - table element name	
addTableCell (string content, isHeader)	Adds cell to 'current row'	* content - cell HTML content * isHeader - '0' - regular cell, '!0' - table header cell	
addContent (string content)	Adds content to document	* content - cell HTML content	
renderDocument (string fname)	Traverses document tree and generates document	* fname - filename for generated HTML document	
string div (string content, string cssClass)	Creates HTML tag	* content - HTML content of tag * cssClass - CSS class for tag	
setProgressBar (string name, value)	Sets progress bar document element	* name - progress bar element name * value - progress bar value (by default should be between 0	

DOCUMENT BUILDER			
FUNCTION			
Function	Description	Parameter [] - optional ^ multiple	Return Value
		and 100)	
setProgressBarCss (string name, string css, string cssBack, string cssFront, string cssText)	Sets CSS classes used by progress bar element	* name - progress bar element name * css - general CSS class * cssBack - CSS for progress bar background * cssFront - CSS for progress bar foreground * cssText - CSS for progress bar value text	
setList(string name)	Sets list document element	* name - list element name	
addListItem (string listName, string content)	Adds item to list element	* listName - list element name * Content - HTML content	
string getLabelContent (string labelName)	Gets string content of label element	* labelName - label element name	Returns string with label content. Returns empty string if no label present with given name.

DATA GRID			
COMMANDS			
Command	Description	Parameter [] - optional ^ multiple	Return Value
createdatagrid	Creates new data grid object. Multiple grid objects can be created within the same environment instance	* name - unique grid name * title - grid window title * dict - path to CSOro dictionary file representing grid data structure	
setdatagridparams	Sets general data grid parameters	* datagrid - data grid object name * [label] - caption label * [fontSize] - data grid global font size * [labelFontSize] - caption label font size	
setdatagridfilters	Sets filter items collection for data grid	* datagrid - data grid object name	

DATA GRID			
COMMANDS			
Command	Description	Parameter [] - optional ^ multiple	Return Value
		* filter^ - list of filter items	
adddatagridcolumn	Adds column to data grid	* datagrid - data grid object name * name - column name (must be unique within table) * [type] - column type [int, float, string] - string by default * [label] - column label * [visible] - column visibility flag [true/false] * [fill] - column fill [true/false]. If true, column will fill as much space in the grid as it can, otherwise the width of column will be determined by its content.	
cleardatagridcolumns	Removes all declared columns in grid	* datagrid - data grid object name	
showdatagrid	Loads data into grid and displays it in a window	* datagrid - data grid object name * datafile - path to CSPro data file with grid data * [filterColumn] - filter column name	Returns case ID of selected grid row. The output is saved to the "DataGrid.tmp" file in the environment temporary folder.

DATA GRID			
FUNCTION			
Function	Description	Parameter [] - optional ^ multiple	Return Value
createDataGrid (string name, string gridTitle, string gridDict)	Creates data grid environment object	* name - data grid name * gridTtitle - title text to display in window header * gridDict - path to CSPro dictionary representing data structure for grid data	
setDataGridFontSize (string dataGrid, fontSize)	Sets data grid font size	* dataGrid - data grid name * fontSize - font size	
setDataGridLabelFontSize (string dataGrid, fontSize)	Sets data grid label font size	* dataGrid - data grid name * fontSize - font size	
setDataGridLabel(string	Sets static text of data	* dataGrid - data grid name	

DATA GRID			
FUNCTION			
Function	Description	Parameter [] - optional ^ multiple	Return Value
dataGrid, string label)	grid label	* label - label text	
setDataGridFilters (string dataGrid, array string filters(), filterNum)	Sets list of filters for data grid control. Filters are displayed in combo box at the top of control and can be selected by user.	* dataGrid - data grid name * filters() - array of filters * number of filters in array	
addDataGridColumn (string dataGrid, string columnName, string columnType, string columnLabel, visible, fill)	Defines and adds data grid column	* dataGrid - data grid name * columnName - column name (must be unique within grid columns list) * columnType - type of column values: string, int, float (string by default) * columnLabel - column header text * visible - '0': column not visible, '!0': column visible (visible by default) * fill - '0': column is autosized to content, '!0' column is stretched to fill the available space (autosized by default)	
showDataGrid (string dataGrid, string dataFile, string filterColumn)	Displays data grid	* dataGrid - data grid name * dataFile - path to data file with grid data * filterColumn - column with values to be used by grid filter (leave blank if no filter present in the grid)	

BROWSER			
COMMANDS			
Command	Description	Parameter [] - optional ^ multiple	Return Value
browse	Opens HTML document inside built-in Web browser	* url - URL to be opened in browser window	

BROWSER FUNCTION			
Function	**Description**	**Parameter [] - optional ^ multiple**	**Return Value**
browse(string url)	Displays document in built-in Web browser	* url - URL of document to display	

DATA COMMANDS			
Command	**Description**	**Parameter [] - optional ^ multiple**	**Return Value**
deletecase	Deletes cases from CSPro data. Evaluates each record in file and removes it if it corresponds to ID values in command parameters.	* datafile - path to input CSPro data file * dict - path to CSPro dictionary describing data file * list of parameters with names of ID Items from dictionary and corresponding ID values	

DATA FUNCTION			
Function	**Description**	**Parameter [] - optional ^ multiple**	**Return Value**
deleteCase (string dataFile, string dict, array string idValues(,), idsNum)	Deletes case from CSPro data file	* dataFile - path to data file * dict - path to CSPro dictionary for data file * idValues(,) - array of data ID values (first dimension is var name, second is var value) * idsNum - number of ID values in idValues array	

Appendix II: Supported System Colors