SHACIRA

User Manual

2i Industrial Informatics GmbH
Wirthstraße 7
D-79110 Freiburg

Tel. +49 (0) 7 61 / 4 52 05-0
Fax +49 (0) 7 61 / 4 52 05-90

e-mail info@2igmbh.de
www.2igmbh.de

1 Overview

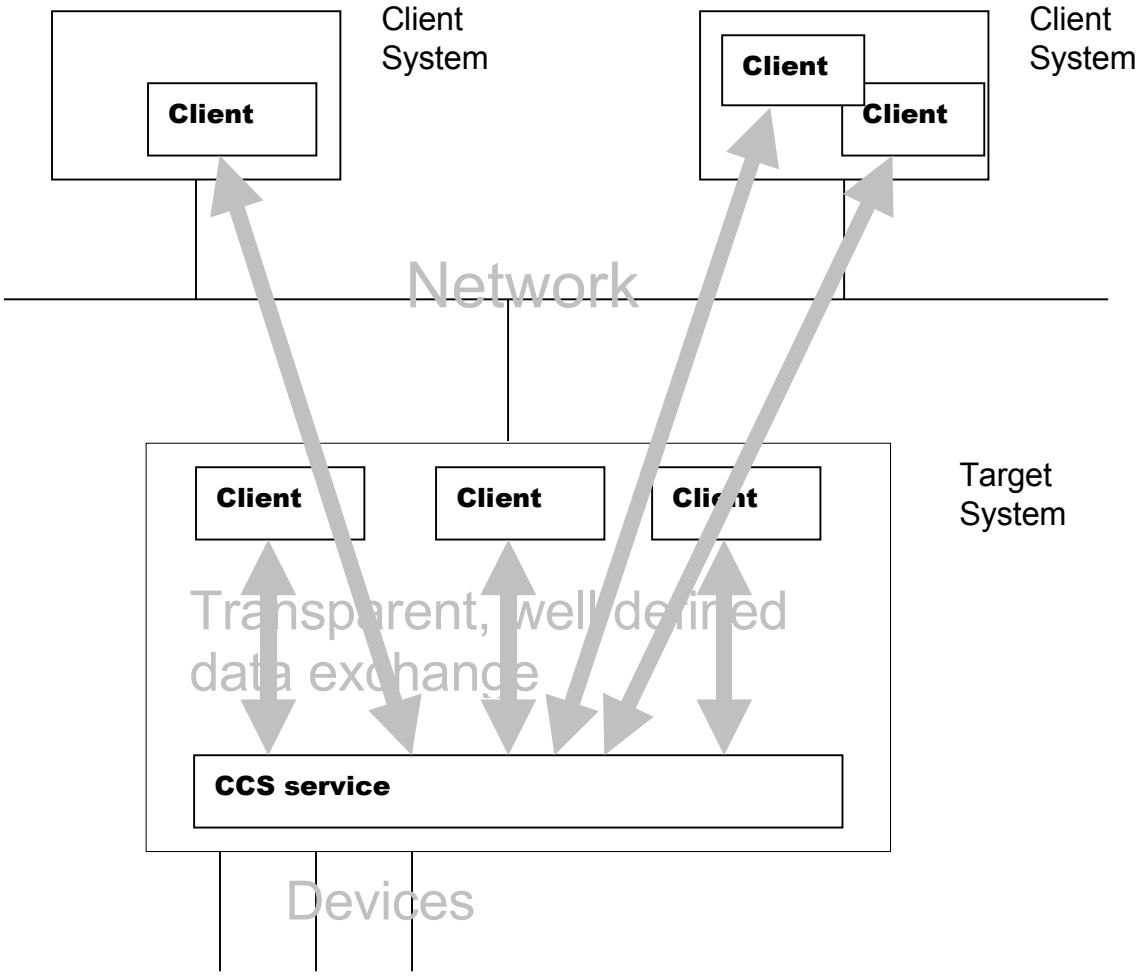


Fig. 1-1 Usage scenario for CCS services

The Shacira Framework consists of two Software components:

1. A CCS Service (**C**ore **C**ontrol **S**ervices)
2. A GUI Framework that is capable of interfacing a CCS Service
   The GUI Framework will be used to develop CCS service clients.

Both components, the CCS service as well as the GUI Framework can be extended in several ways to fulfil application specific requirements.

2User Functions

User functions are a general way to realize application specific functionality within the Shacira Framework.

User functions must be declared through a signature similar to C function declarations. User function arguments are restricted in two ways:

- user function arguments generally are input arguments
- user function arguments are restricted to Shacira Base Types and Shacira String Types

There is no restriction on the argument count.

User functions that extend the Shacira GUI Framework are referred to as GUI functions. User functions that extend CCS services are referred to as Model functions.

Every user function is associated with a function class that implies the usage context of the function.

2.1Function classes

Function classes are split up into the 2 groups of Model functions and GUI functions:

Model function classes

- Filter functions
- Conversion functions
- Limit functions
- Unit functions

GUI function classes

- Dark (blanking) functions
- Plausibility functions
- User functions
- Button functions
- Signal filter functions
- Slot functions

Additionally to the declared input arguments, every function class has so called automatic arguments. Automatic arguments are arguments that are generally supplied when calling the function. Automatic arguments must not be declared and precede the declared argument list. Count and semantics of automatic arguments vary depending on the function class.

Details to user function usage follow in the chapters about model functions and about GUI functions.

2.2Function References

Function references can be compared with C-Function calls. Function references allow embedding of function calls into code that is not C-code. Embedded function calls can be placed at specific places to hook application specific functionality.

- Text type properties of Qt Widgets used in the Qt Designer (explained later)
- Specific properties of the data model

Afunction reference will be interpreted and called by the Shacira Runtime System. Nevertheless, the implementation of the function is ANSI-C.

3CCS Service

When we talk about the CCS service, a process that hosts a couple of **C**ore **C**ontrol **S**ervices is meant. A CCS Service consists of a couple of different services. This services build a runtime system that generates application data and propagates this data up to a specific user interface. Data exchange between a user interface and a CCS Service comes in two forms.

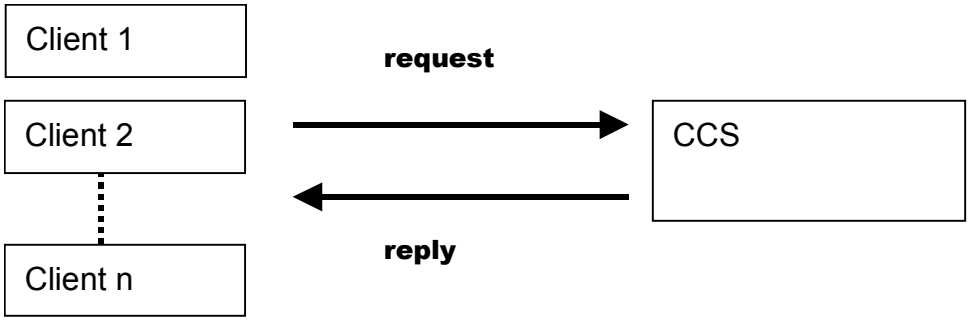
1. The asynchronous way:
   CCS services generate events to inform other components of the application, for example the user interface, that "something" has happened.
2. The synchronous or client-server interface that offers query methods to acces CCS data.

3.1Basic CCS Communication Principles

Throughout the architecture of a CCS service, two basic communication mechanisms are used. These basic principles are used two implement the client Interface of a CCS service. A CCS service offers two logical interface types:

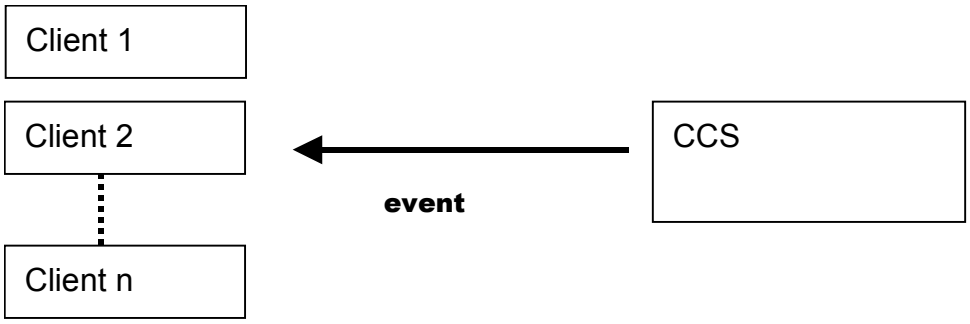
1. Synchronous communication
2. Asynchronous communication

Synchronous Communication



CCS Data Services are the client server interface of a CCS service. If a client needs some information from the CCS service, this information must be requested from the CCS. A response (even an error condition) is delivered synchronous. This mechanism guarantees to get a reply.

Async* chronous *Communication



The asynchronous communication part can be viewed as a couple of channels that carry information as a sequence of events appearing over time. Events are available as event objects delivered over a specific channel called event channel.

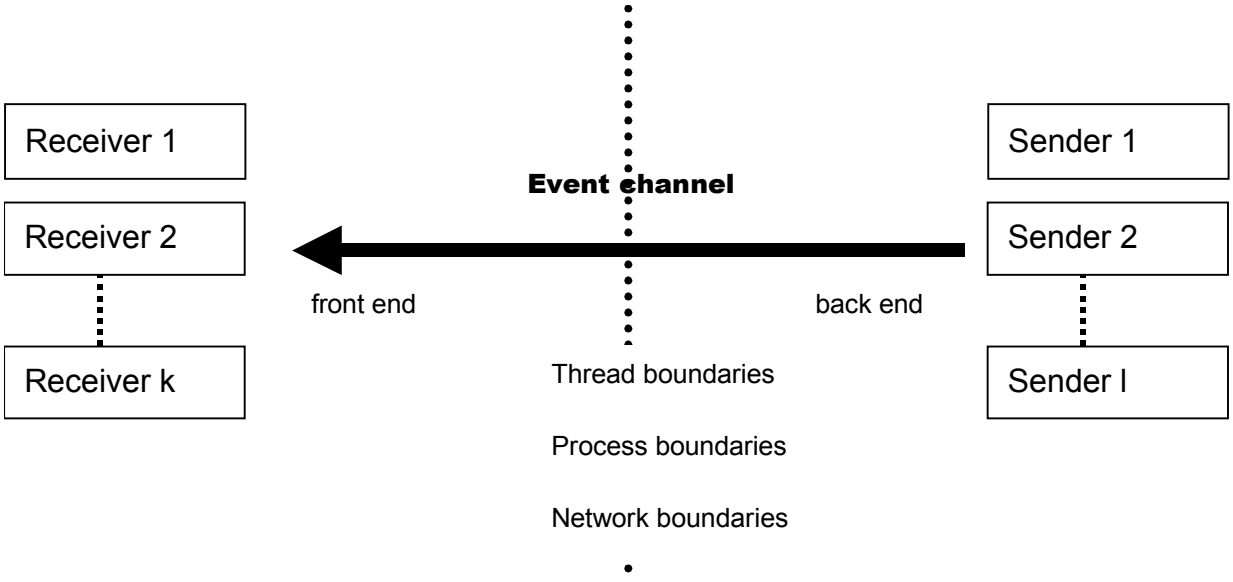
CCS event objects are called transient objects. They all inherit from the base class cTransientObject. CCS event objects - shortly called transient objects – are used to communicate arbitrary informations between different application components. Transient objects must implement a serialization interface. Serialization of transient objects offer the possibility to send objects over process and network boundaries.

Every interested application component is able to register with the end of an event channel to receive events in the form of event objects. Sources for events in a CCS Service are device objects, program objects and other services. A typical situation is to register for data change notifications of a variable. The deliverance of events is not synchronised with the client. There is no guarantee when or if at all an event is delivered.

In contrast to a simple callback mechanism, the CCS event interface is not coupled directly to a client. This construction avoids the possibility of negative side effects by directly calling client functions. Event delivery can neither delay nor crash CCS service functions.

The asynchronous communication interface of a CCS service is based on a more general communication mechanism used throughout a CCS service: **Event Channels**

Event Channels



The event channel model must conform to the following requirements:

1. An event channel has a front end to receive data
2. An event channel has a back end to send data
3. An event channel can be used by arbitrary channel clients to send data (senders)
4. An event channel can be used by arbitrary channel clients to receive (listen to) data (receivers)
5. The number of senders is not restricted nor is the number of receivers
6. Every event sent by a sender to the front end of the event channel is transmitted to every receiver listening to the back end of the event channel.

Event channels are not restricted to transmit data within processes. As a consequence the CCS system offers different event channel implementations. Event channel implementations that are used within the CCS service are event channels that propagate events with minimal time or performance consuming overhead. These event channels are very fast but restricted to in process usage. Receivers and senders can only be threads within the same process. These very fast event channel implementations can be used for the communication between clients and CCS service when the CCS service is housed within the client process.

There are further event channel implementations that offer the opportunity to propagate events across process and even network boundaries.

Actually there are two event channel implementations that are able to cross process and network boundaries based on similar technologies:

1. An implementation based on the CORBA event service specification.
2. An implementation on top of the CORBA synchronous object interface.

3. An implementation based on the TCP/IP UDP protocol using datagrams.

The selection of a specific channel implementation can be configured using the CCS service configuration interface.

3.2CCS Logistic Infrastructure

Every CCS service contains a definable infrastructure that manages propagation of event objects. For this purpose we use defined and free configurable queues called logistic queues. Logistic queues are specific channels that are capable to buffer a certain amount of transient objects. Logistic queues block if the the queue is full.

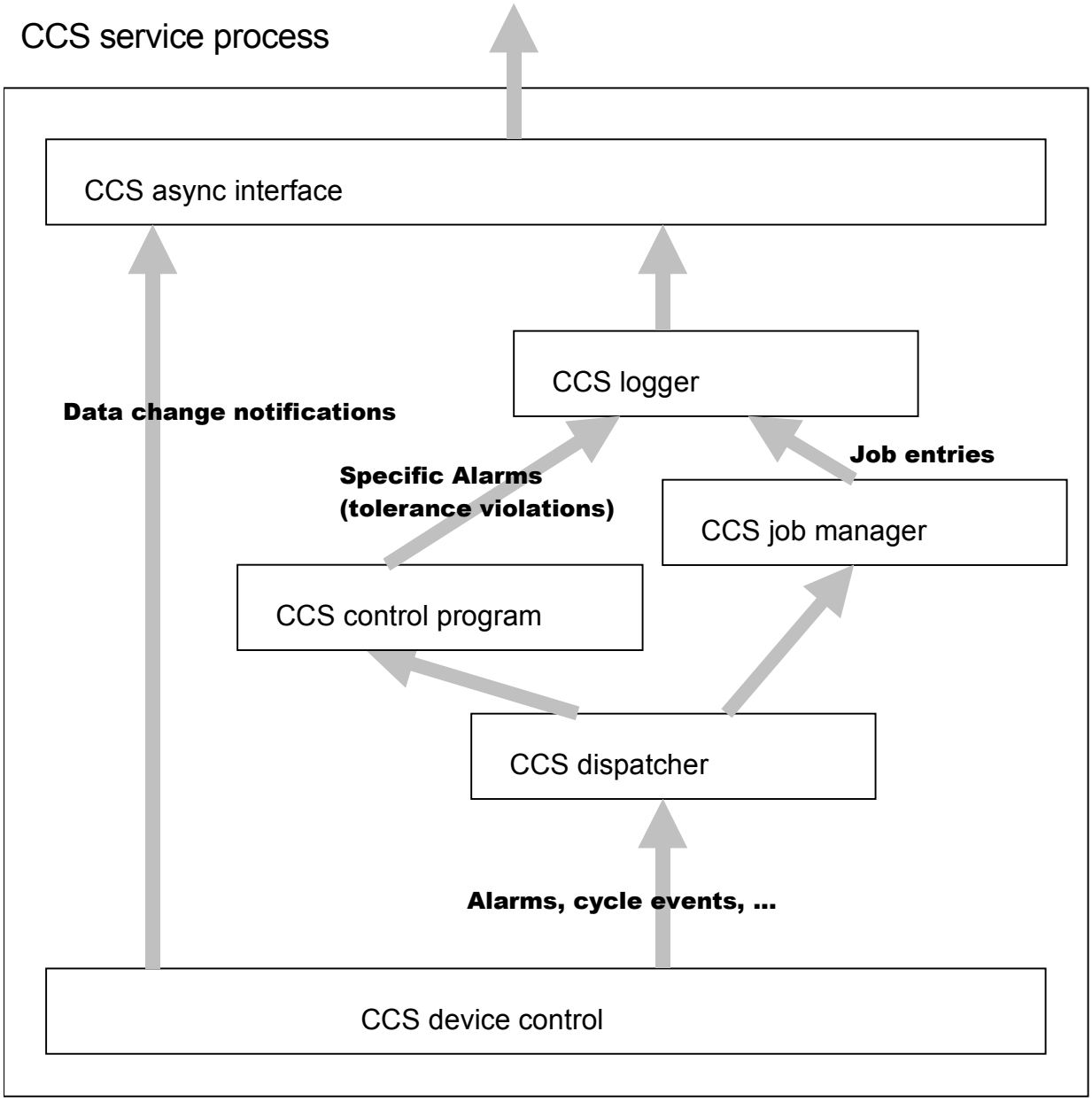


Fig. 3-1 Example scenario of CCS logistic queues

3.3CCS Software Layers

A CCS service consists of a couple of well defined software layers. Every layer fulfills specific tasks.

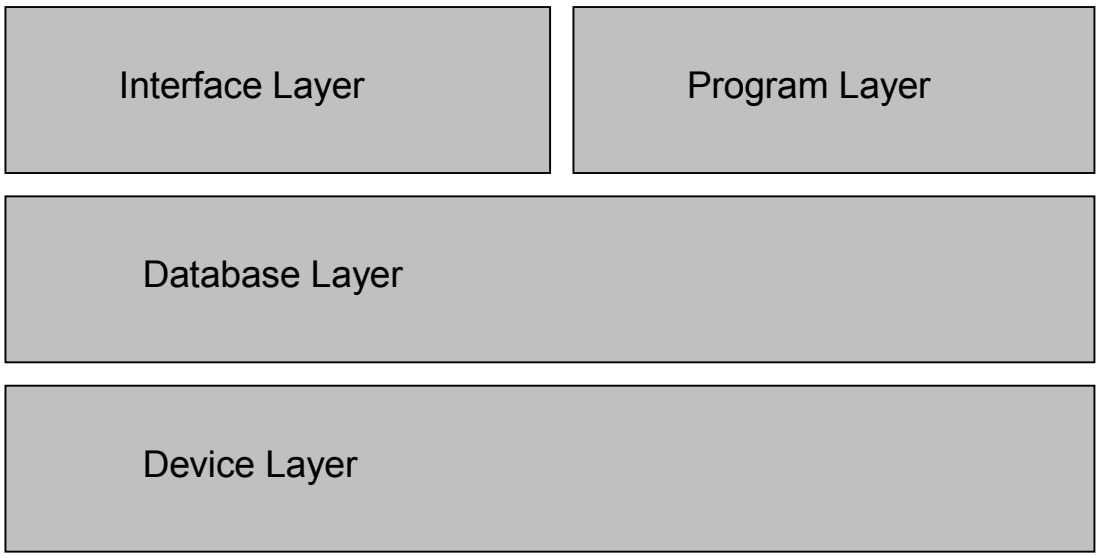


Fig. 3-2 CCS Software Layers

- The **Interface Layer** enables application developers to show and record application data that is supplied and manipulated through the lower layers.
- The **Program Layer** offers the application programmer services related to time constrained tasks that are not directly obvious to the user interface. Nevertheless the Interface Layer can directly reflect state changes of application data if needed.
- The **Database Layer** supplies services for the higher layers through well defined variables. Variable descriptions reside in the data model of the application. The database layer also manages resources that are needed to meet the application developers intention.
- The **Device Layer** holds abstractions of arbitrary and free programmable device interfaces this layer enables the higher layers to exchange data with concrete devices of the process to be controlled and visualized.

CCS software layers are designed to interface only to layers that have direct contact in the layer diagram above. This rule forbids to directly communicate to devices from the Interface or the Program Layer.

3.4CCS Database

The CCS database supplies and manages resources that are used to store and retrieve data.

The CCS database is a specific "database" that holds application specific data. The CCS database interface is directed to the specific needs of holding data in a manufacturing environment and is not comparable to known database technology interfaces (the CCS database is for example no SQL server). Nevertheless this storage simply is referred to as the CCS database.

The description of a CCS database is done using a specific language SLANG (**S**hacira **LANG**uage) to model structure and further properties of the data that is used by the application.

3.5Device abstraction

To interface with peripheral devices such as PLCs CCS offers device objects to exchange data with devices in a common and well defined way.

3.6Programs (tasks)

Programs or tasks are offered to do cyclic work that not originates from atomic state changes in the process environment.

3.7Persistence

To persist data over shutdown periods of the system, a persistence service is implemented.

3.8Storage

To exchange data among different CCS applications, a storage service is implemented that stores data of the CCS database into flat files.

These files then can be imported into other CCS applications.

3.9Data Modelling in SLANG

The abstract syntax of the modelling language can be found in:

ShaciraLanguage.html

The abstract syntax shows the structure of the underlying language. Syntactical notations are expelled in C-Style comments if necessary.

Semantic description

A Shacira model can contain a set of definitions. The order of the definitions is arbitrary with the execption of references. When for example a function is referenced the declaration for the referenced function must already exist (function declarations must precede function references). The same is true when variable references are used (the variable definition must precede every reference of this variable) and when unit definitions are used.

Database Definition (DatabaseDef)

```
database "plc_vars" device=SPS1 {
  variable definitions ...
```

};

A database definition contains the description of a couple of data items called variables. Variables can be structured up to 4 dimesions.

When a device is specified for a database using the specification device=<device name> then all mapping specifications within this database are directed to the device with the specified name.

The semantics of a database allows the usage of more than one database with the same name. The database name is semantically simply a property of every variable definition within this database.

Function Declaration (FuncDecl)

A function declaration defines a function signature similar to C-function prototypes Function argument types are restricted to the SHACIRA base and string data types.

When a function reference (function call) is used throughout the runtime system, the function declaration is required to supply the proper arguments and to return the correct value type.

When a function reference (function call) is part of the model, the function declaration is required to implement type checking at compile time.

conv long fact(long number);

is an example for the declaration of the factorial function as a conversion function.

Function Reference (FuncRef)

A function reference defines a function call with concrete arguments.

The arguments that can be supplied recruit from the following sources:

1. A constant value of type BaseType or StringType compatible to the specific argument declaration.

   example: fact(200)

2. A variable reference. The variable data type must be compatible to the specific argument declaration.

   example: fact(array[50]) array[50] here is a variable reference

3. Another function reference. The return type of the referenced function must be compatible to the specific argument declaration.

   example: fact(fact(3)) fact(3) here is the function reference

Variable Definition (VarDef)

A variable definition describes a variable of the data model. Variable name, structure and type must be specified. The description of a variable can be extended by specifiying properties directing to the areas:

- Verbose textual descriptions
- User function references for conversion, filtering, access and range control
- Mapping to a device
- Physical Unit description
- Persistence
- Dataset management
- …

Example:

```
SetStrPlstShotSize[InjUnits] // mm, Achtung maximum,Index 2 sollte Aggregat verwendet werden
  float
  vartype=set
  unit=StrVol
  description="Stroke plasticising"
  persist=file
  file=Tool
  minimum=0
  maximum=MaxStrPlstShotSize()
  map register ushort(1) {ssdos1, -1, -1, -1, -1, -1}
;
```

A variable definition defines all properties of a variable. This includes type and structure specifications and the mapping of the variable to a device. When a variable is not mapped to a device the variable is called an internal variable. Mapped variables are actually mapped to the device specified in the Device Specification of the database.

A variable definition starts with a variable name that is a legal Identifier specified by the token specification Ide.

Structure information is specified using an array specification with up to 4 dimensions. Structure information is immediately supplied behind the variable name. A dimension is specified by an Integer value (dimension size) embraced by [].

When no structure information is supplied the variable is a sc

array[100] defines a variable with one dimension of size 100.

Behind variable name and structure information a data type for the variable must be specified. Data types are restricted to the following data types:

Shacira base types:

- bool: 8 bit char data (contains 1 if true and 0 if false). When setting a bool value a value != 0 will set the bool variable, 0 will reset the bool variable.
- char: 8 bit signed character variable ranging from –127 to 127.
- uchar: 8 bit unsigned char variable ranging from 0 to 255.

- short: 16 bit signed variable ranging from –32767 to 32767
- ushort: 16 bit unsigned variable ranging from 0 to 65535
- long: 32 bit signed variable
- ulong: 32 bit unsigned variable
- float: 4 byte single precision variable
- double: 8 byte double precision variable

Shacira string types:

- string(size): fixed size string variable (size is supplied as parameter)
- wstring(size): fixed size wide string variable (size is supplied as parameter)
- byte(size): fixed size byte string (size is supplied as parameter)
- object: variable size byte string (pointer to memory). The supplier determines the size.

Types bit8, bit16 and bit32 should only be used in the description of device mappings. They are not used to describe CCS data.

Variable related properties

Besides structure and type information, a variable has properties that may affect functional semantics or localized variable description:

Description (description="description")

Example:

description="Stroke plasticising"

A description is an arbitrary string that describes the variable. The description of the variable can be seen as comment of the developer.

Text (text="text")

Example:

text="Plastifizierweg"

The variable text is a short description of the variable that will be localized and translated. The text of the variable therefore can be used to generate localized representations to denote a variable and that can be also understood by machine operators.

Dim1 (dim1text="text")

Example:

dim1text="Aggregat"

Dim2 (dim2text="text")
Dim3 (dim3text="text")
Dim4 (dim4text="text")

Dim1 to Dim4 assign names to dimensions of the variable. These texts are also localised and can be used to show users localised and better understandable names of variables.

VarType (vartype={act,set,volatile})

Example:

vartype=set

The variable type describes the usage and the semantics of Variables that are mapped to devices.

act

The variable is an actual value. It is not possible to set the value of a variable that is specified as act. Act values are refreshed by the device cache. The device cache is used to optimize transfer operations for act values to avoid unnecessary traffic between a device and a Shacira application. When somewhere within the application an act value is queried, this value is read from cache (not from the device).

set

Variables of type set are variables that are managed only within the CCS service. Mapped devices hold a value that is identical to the value within the CCS database. Therefore the CCS Service is responsible to transmit set values to the device whenever the value in the database changes or the service starts up.

To optimize startup behaviour device objects can buffer set values and transmit all data in a single download step. The values of set variables are guaranteed to be synced with the internal CCS database if a device has the possibility to signal a download request.

When the device signals a download request after loss of data all data will be transmitted to the device.

volatile (default)

Volatile variables exchange data with a mapped device when the variable data changes (a device write operation will be initiated) or when the value is read (a device read will be initiated). Neither caching nor buffering is possible.

PersistenceType (persistence={ram,bbram,file})

Example:

persist=file

The persistence type of a variable specifies the storage location for the variable data. This specification determines the life cycle of variable data. The data of bbram and file based variables persist process and even system shutdowns, ram based variables do not and must be initialised whenever a CCS service restarts.

RefreshType (refresh=<milliseconds>)

Example:

refresh=100

RefreshType (refresh=event <event type>)

Example:

refresh=event cycle

RefreshType (refresh=event <refresh code>)

Example:

refresh=event 2

The refresh type of a variable with vartype = act determines the moment when data is refreshed from a device. The most typical usage is to specify a refresh as integer value that defines a refresh cycle. This value is interpreted as milliseconds.

Beneath cyclic refresh specifications it is possible to bind a refresh operation for an act variable to system specific (second form) or application specific (third form) events.

File (file=Identifier)

Example:

file=Tool

Variables of variable type set can be stored in a file that can be written to changeable media. This file is organized as a couple of subfiles. The file specification associates a variable with one of these subfiles.

FilterFunc (filter=<FuncRef>

Example:

filter=WDChangedTime()

A filter function is a function that is used to set and get variable values. It filters all of the standard behaviour when getting and setting variable values. The filter function determines what happens if a variable value is written or read.

The function class of the referenced function must be filter.

MinSpec (minimum=<Agument>

Example:

minimum=MinNumHeatZones() *or*
minimum=0

The min specification determines the lower limit for the variable values. This limit can be specified as an argument that means it can be specified as a function reference, as a variable reference or as a constant value.

MaxSpec (maximum=<Agument>

Example:

```
maximum=MaxNumHeatZones()      or
maximum=22
```

Same as MinSpec but specifies the upper limit for variable values.

3.10Variable mapping to a device

Every variable defined in a CCS data model can be mapped to a device. The variable then represents device data that can be buffered, cached or transformed.

A device mapping description comes in two forms:

- Name based mapping
- Address based mapping

These forms of the mapping specification are contributions to the addressing scheme of devices that are based on either names or integer addresses.

To simplify mapping notations for structured variables it is possible to specify a list of mapped items .

Scalar address based mapping (map buf-spec type address:bitpos)

Example:

```
ActTimPlstCool
.
.
  ushort(1)   map register ushort 0x55
```

This definition maps a device address to an unstructured variable.

Scalar name based mapping (map buf-spec type name)

Example:

```
vidCam1Source byte(443000)
.
.
  map Channel0 byte(443000) "Image";
```

This definition maps a device name to an unstructured variable.

Structured address based mapping (map buf-spec type {address:bitpos,...})

Example:

```
SetStrPlstShotSize[6]
.
.
  map register ushort {0x68, -1, -1, -1, -1, -1}
```

This definition maps device addresses to a structured variable in the form of a flat address list.

Structured name based mapping (map buf-spec type {name,...})

Example:

Example:

```
VidCamSource[2] byte(443000)
.
.
  map Channel0 byte(443000) {"Image1", "Image2"};
```

This definition maps device names to a structured variable in the form of a flat name list.

Buffer specifier

The buffer specifier is an arbitrary identifier that is interpreted by the concrete device the variable is mapped to. Using buffer specifiers enables structuring of device address spaces or data areas.

Mapped data type

The mapped data type is the device data type that must be mapped to the variable data type. Concordance of the two data types is not required. The necessary conversions are done by the CCS service (if possible).

Conversion function ("conversion" "=" FuncRef)

Example:

```
  conversion=MultFloat(10)
```

A conversion function reference can be supplied to realise transformations to variable values on their way from and to the device. The function class of the function reference must be conversion.

Extensions for address based mapping

When address based mapping is used there is a possibility to map bits or even bit ranges rather than bytes words or double words to arbitrary variables. This can be done by specifiying a bit position or a bit range to the address separated through colon.

bit1 bool map short {5:1}
maps bit no 1 of word 5 to a bool variable

bit1 uchar map short {5:0-3}
maps bit no 0,1,2 of word 5 to an unsigned char value

Note: the organisation of bit and byte orders is left to the specific device interpreting the notation. By default Intel bit and byte orders are used. Every device object can override this order using a specific bit operator class that inherits from the class

cBitOperator.

Auto Address Increment

When mapping device data to structured variables, the required address list may not be complete. Missing specifications are added automatically using an auto increment mechanism:

Array[4] short map short {0}

is the sane as

Array[4] short map short {0, 1, 2, 3}

Note: this mechanism is used and works only for address based mappings because an auto increment can not be defined for names.

3.11Extending a CCS Service

There are 3 methods to extended CCS service functionality.

1. Adding user functions to the data model.
2. Adding application specific programs to the CCS service.
3. Adding application specific devices to the CCS service.

Functions that extend a CCS service

Custom specific hook functions are used as CCS extensions referred to in the data model of a CCS service or as GUI extensions that must be declared in a declaration file.

A user function must be associated with a function class. The function class preceeds the function declaration and implies the semantic and circumstances of function execution.

User functions that are hooked into a CCS service are sometimes called model functions because the their function declarations and references are part of the data model.

Filter functions (class filter)

Filter functions act as general purpose data providers for variables. They supply a value when a variable is read and they execute some predefined action when a variable is written. The return type of filter functions must be compatible to the data type of the variable associated with.

Conversion functions (class conversion)

Conversion functions transform values on their way from and to devices. They are associated with the mapping of a variable to a device. The value to be transformed is supplied as first argument to the function. This is an automatic argument and must not be declared within the function declaration. The return type of conversion functions must be compatible to the data type of the variable associated with.

Limit functions (class limit)

Limit functions supply a minimum and a maximum value for variable input. The return type of limit functions must be compatible to the data type of the variable associated with.

Unit functions (class unit)

Unit functions implement conversions between different physical units of a variable. The return type of unit functions must be compatible to the data type of the variable associated with.

Access functions (class unit)

Access functions implement access from host interfaces. Conceptual an access function is similar to a filter function but restricts to the access over a host interface.

Using application specific programs tot extend CCS service functionality

An application specific program inherits from the base class cProgram. Task of a program is to do application specific work that must be done in background. A typical application ist to control or record variable values.

3.12Databases Contexts and Variables

To access CCS data a Shacira application uses the Set- and Get-Methods of a variable object.

Variable objects are C++-objects of the class cVariable a variable can be identified through the variable name. A variable object must be queried from a CCS database that is represent by the so called context object.

A context object is a C++-object of type cContext. This context object houses all variables that are modelled in the corresponding. Data model. A context object subsumes access to CCS services and access to variables in the CCS database.

All variables that live in a CCS database can be accessed using the method cContext::Variable(CONST_STRING_T var_name). This method returns a pointer to cVariable type object that offers an API for getting and setting values of the variable. Beneath getting and setting values a cVariable type object offers methods to retrieve variable state information and variable properties such as data type, precision.

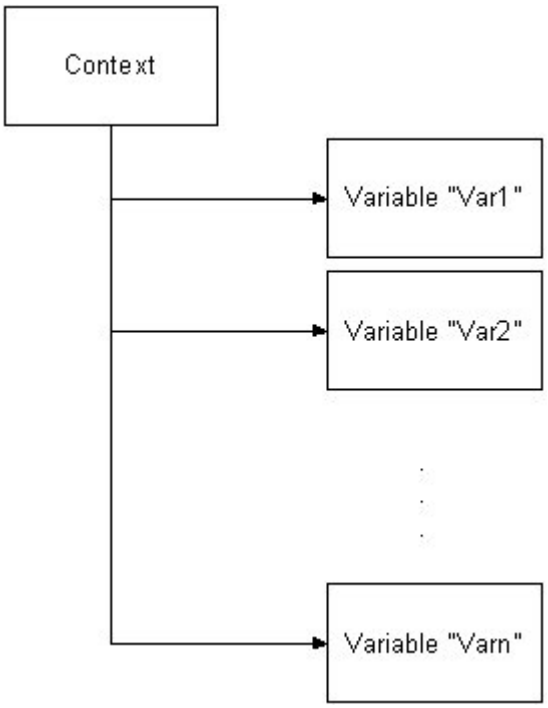


Fig. 3-2 Context structure of a CCS database

3.13Variables

Before handling CCS device aspects it is important to understand the central term of "variable types".

A variable type influences the interaction between variables and devices in a crucial way. Actually a variable can be assigned to three different variable types:

1. A volatile variable (vartype = volatile) this is the default, when no vartype is specified
2. A set varibale (vartype = set)
3. An act variable (vartype = act)

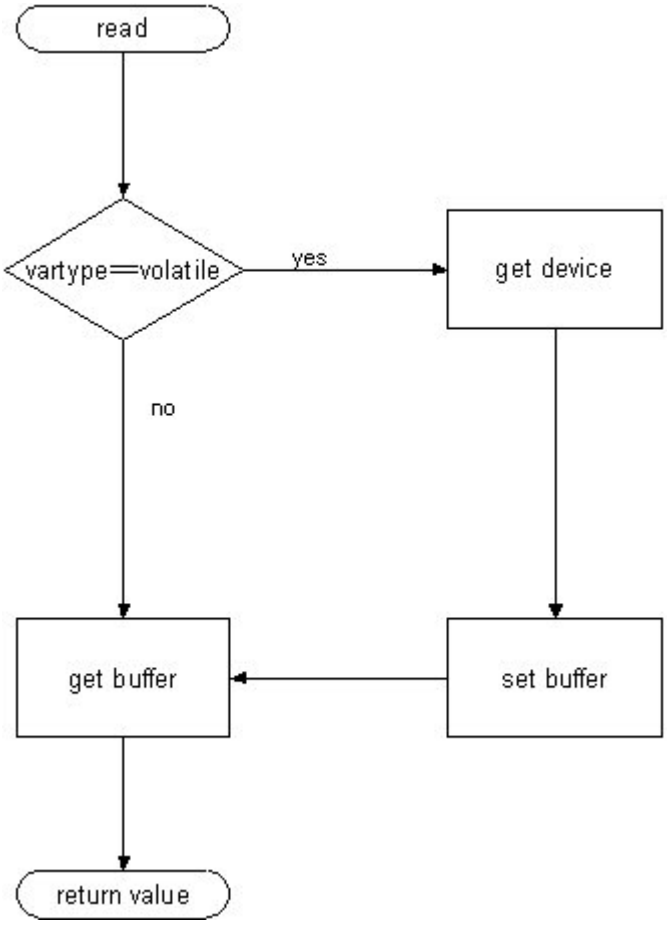


Fig. 3-3 Variable read operation

Read operations on a variable access a device value only when they are of type volatile. In the case of set and act type variables values are read from the internal buffer.

The values of act type variables are refreshed by the device cache, values of set type variables are communicated only in one direction from the internal buffer to the device.

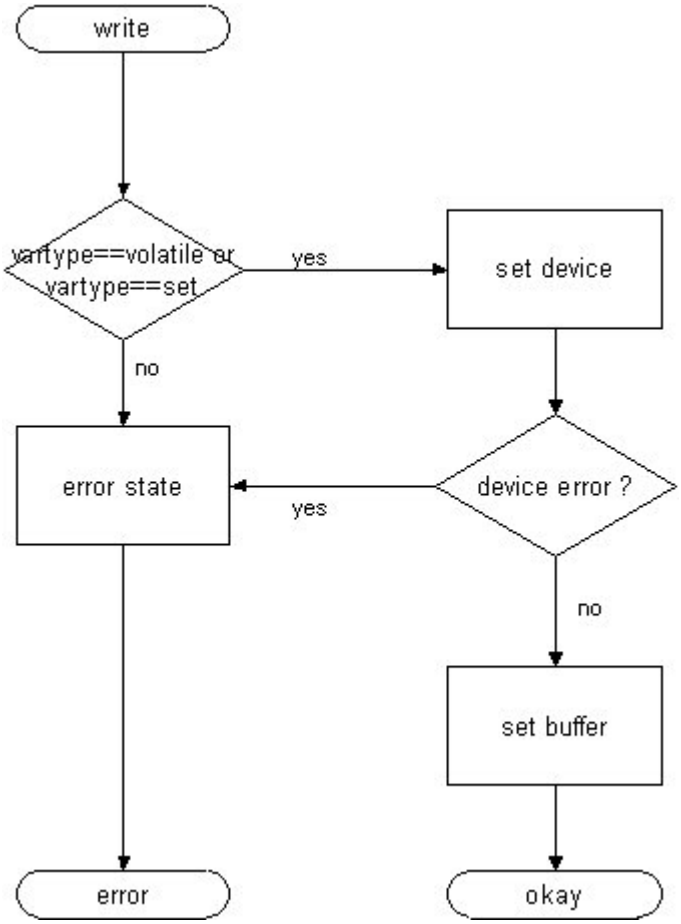


Fig. 3-4 Variable write operation

Write operations on a variable set a device value only when they are of type volatile or set. Trying to write to an act type variable generally leads to an error situation.

When setting the value of a set or volatile type variable, CCS first tries to set the value on the device and refreshes the internal buffer only when this operation succeeds.

4How to Implement Application Specific Devices

What is a device or device object?

Every variable that is mapped to a specific device uses the device object to exchange data with this device. For this purpose a device object has an abstract device independent interface to perform this task.

When exchanging data with a device only the methods of the abstract interface are used to exchange data by the variable.

What is needed is a sort of addressing scheme that can be referred to in the data model. Shacira CCS offers a 2 level addressing scheme:

1. First level addressing using a buffer specifier that is a string
2. Second level addressing using either a numeric address (address based mapping) or a name (name based mapping).

The second level addressing allows mixing together address and name based mapping, thus mixing together both addressing types in one device object is not recommended. In general the protocol to access device data poses a natural addressing type to the device object implementation. For example PPCCOM and Siemens S7 are typical address based, while OPC is typical name based.

The interpretation of both addressing schemes is left to the concrete device object implementation. The concrete addressing scheme is transparent to the base system.

The first level addressing scheme using a buffer specifier is intended to address different data areas, for example registers and logic words in the case of PPCCOM. Data throughput considerations can affect the design as much as ease of implementation or ease of usage when modelling device data.

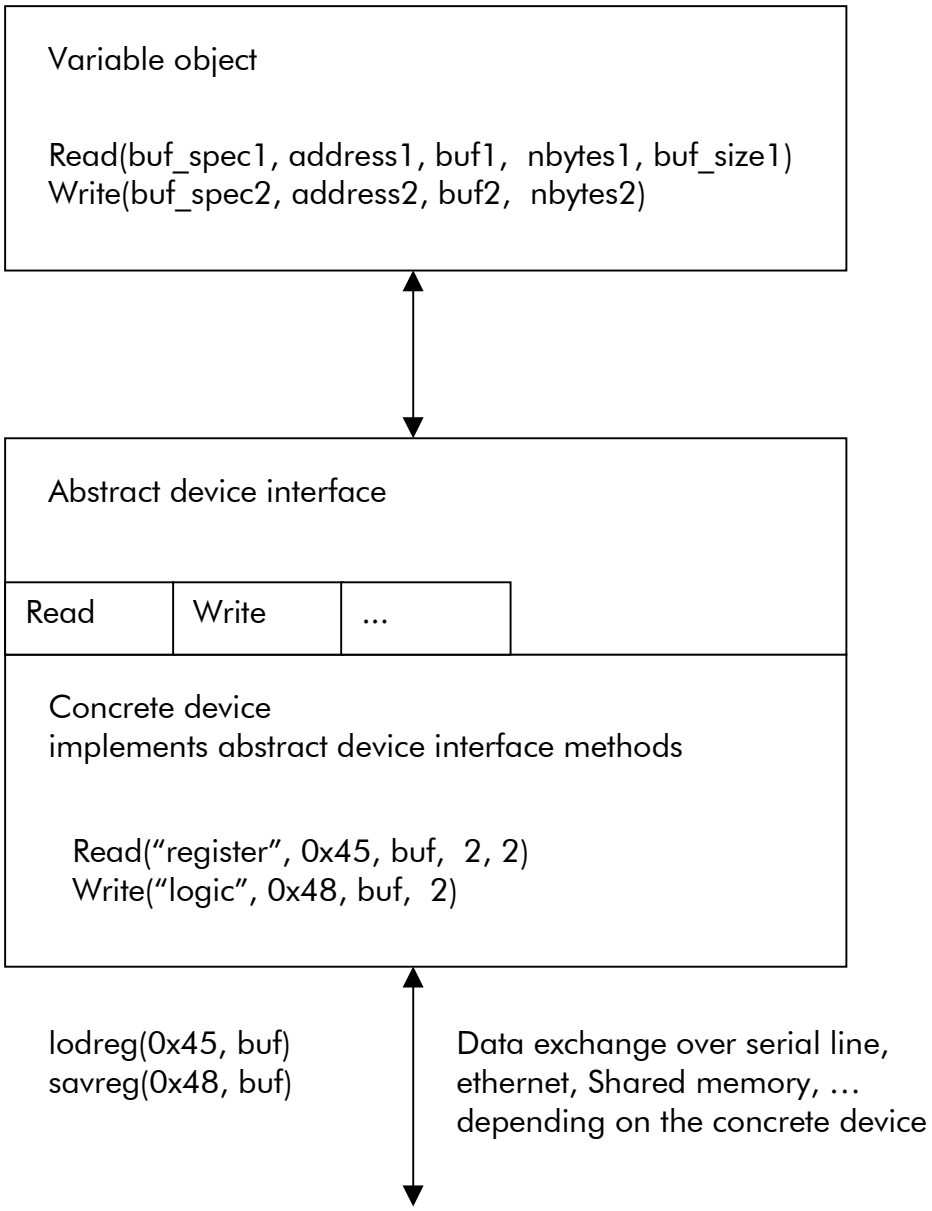


Fig. 4.1 Data exchange using a device object

When implementing a new device a device programmer designs a new class that inherits from cDevice, the base class for all devices.

The minimum interface a new device must implement are the two methods device::Read(…) and device::Write(). This are the basic methods for exchanging data with a device. The getter and setter methods device::Get(…) and device::Set(…) must be reimplemented, when the device should support direct reading and writing of Shacira base data types or strings.

The methods device::Cache(…) and device::Buffer(…) should be reimplemented, when the device supports caching.

4.1Role of buf_spec, var_name and address arguments

4.2Bit Operators

Bit operators are used to interprete values of Shacira base types like long, ulong, float, … to requirements originating from processor architectures of specific devices. It addresses byte and bit orders of numeric datatypes.

4.3Device caching

Th task of a device cache is to maintain the actuality of act variable values as required (specified in the data model).

To enable device specific caching strategies device specific caches are a means of CCS service extensions. When an application (device) specific is needed the implemented class must inherit from the cache base class cCache.

Some more general cache classes are already implemented within the Shacira CCS Framework including an optmized address based cache (cBlockCache) and a simple name based cache (cStringCache).

5How to Implement Model functions

The first step in implementing a model function is to declare this function in a data model. The function declaration must precede every usage (function reference) of this function.

```
filter long  filter1(long arg1);
```

The above function declaration declares the filter function filter1 that has one long type argument called arg1.

When declared the function can be referenced later on in the data model:

```
..
filtered_var long
 description="this is a filtered variable"
filter=filter1(27);
```

The function reference filter(27) is similar to a function call where concrete arguments (27 in this case) are supplied. Translating the data model with the model compiler mdlc adds a C-function prototype to the generated prototype file:

```
LONG_T filter1(cContext * _context, LONG_T _value, ULONG_T _flags, LONG_T _i1,
                LONG_T _i2, LONG_T _i3, LONG_T _i4, LONG_T arg1);
```

The C-function signature is not the one that we would expect. We declared only one argument called arg1. But this argument is the last one in the argument list and there are a couple of arguments that precede the declared argument. The preceding arguments are so called arguments that are supplied to the function. Count and type of automatic arguments differ among function classes.

_context is a pointer to the context the variable belongs to (the variable that contains the filter function reference). _context enables to access informations that can be supplied by this context; this includes accessing values from other variables.

_value is the current value of the variable at the time, when the function is executed.

_flags codes some information about the communication direction, when the function is called.

_i1 … _i4 are the indices which denote the position the filter function is currently associated with.

As a second example consider the following max function:

```
limit long  MaxIndex(long which);
```

5.1Automatic arguments of model functions

Function class	Parameter	Parameter type	semantics
Filter	_context	CContext *	Pointer to CCS context
	_i1, _i2, _i3, _i4	Long	Indices of the selected data element
	_value	Same as return type	Actual value
	_flags	Unsigned long	Flags to control function execution
Conversion	_context	CContext *	Pointer to CCS context
	_i1, _i2, _i3, _i4	Long	Indices of the selected data element
	_value	Same as return type	Value to be converted
	_flags	Unsigned long	Flags to control function execution
Limit	_context	cContext *	Pointer to CCS context
	_i1, _i2, _i3, _i4	long	Indices of the selected data element
Unit	_context	cContext *	Pointer to CCS context
	_value	Same as return type	Value to be converted
	_flags	Unsigned long	Flags to control function execution
	_i1, _i2, _i3, _i4	long	Indices of the selected data element

Tab. 4-1 Automatic arguments of GUI function classes

6How to Implement Application Specific Programs

To implement an application specific program the application programmer has to create a new class that inherits from cProgram which is the base class for all programs running in a CCS service,

A program or task is technically spoken a thread that runs in background. A program will be "cycled" within a definable interval (default 500 ms). When the program is cycled the virtual method Cycle() of the cProgram base class is called. To hook into the program cycle the application programmer has to reimplement the Cycle method.

When the program starts running as a first step the virtual method Initialize() is called. This method can be overloaded by the application programmer to do initializing stuff.

Beneath the Cycle method that is executed by the thread there are two further methods that can be overloaded to extend program functionality.

1. ProcessEvent(cTransientObject * object)
2. ExecuteCommand(ULONG_T command)

ProcessEvent is executed when the program receives a CCS event (transient object). This method is executed asynchronously to the Cycle method.

ExecuteCommand can be used to communicate synchronously from the user interface with a program to activate some special functionality.

7Variable References

Before explaining the GUI Framework integration, an often used and often misunderstood term must be clarified to understand the Shacira Framework concept. Understanding the concept of variable reference is vital to integrate CCS data access into a GUI framework. A variable reference thus is no GUI specific construction but it is one of the major concepts to integrate a user interface with a CCS database.

For this purpose lets have a look on a two dimensional variable named "matrix1" that lives in a CCS database:

```
database demo device=device1 {

   matrix1[5][10] float(2)
    vartype=set
    description="varioable definition for demonstration purpose"
    text="matrix1"
    dim1text="index 1"
    dim2text="index 2"
    persist=file;

};
```

To access data of this variable there is the known possibility to retrieve a pointer to the variable object using the context:

```
cVariable  * variable = context->Variable("matrix1");
if (variable != NULL) {
  /// variable exists within the database represented by context
  FLOAT_T value = 0;
  variable->Get(value, 3, 2, -1, -1, 0);
  /// now holds the value of variable matrix1, at position [3][2]
}

cVariable  * variable = context->Variable("matrix1");
if (variable != NULL) {
  /// variable exists within the database represented by context
  variable->Set(10.5, 3, 2, -1, -1, 0);
  /// now the value of variable matrix1 at position [3][2] is set to 10.5
}
```

This method works fine and can be applied elsewhere in C++ application code.

The variable reference concept is an extension to this mechanism and is implemented on top of the cVariable class (thus this is not transparent to the application programmer).

When using cVariable to retrieve a variable value an application programmer has to retrieve a pointer to the desired variable and then gets the desired value by calling the Get-method supplying the needed arguments to address position [2][3].

The variable reference approach integrates these two steps in one step.

A variable reference can be denoted by the following specification:

```
Matrix1[2]][3]
```

Part of the API of a cContext object is the Method cContext::VarRef(CONST_STRING_T var_spec). The VarRef method supplies a pointer to an object of class cVarRef. cVarRef is the C++-class that holds a variable reference. The variable reference has an API that is similar to the cVariable API. Methods to set and get values of a variable reference have a differ in signature from the cVariable versions.

To get a value from a variable reference the application programmer has to call GetValue instead of Get and SetValue instead of Set. The cVarRef signatures of the getter and setter methods do not incorporate structure information (indices), because a varriable reference addreses exactly one variable position when calling GetValue or SetValue.

The following code snippets show the above examples but using a cVarRef object instead of a cVariable object.

```
cVarRef  * var_ref = context->VarRef("matrix1[2][3]");
if (var_ref != NULL) {
  /// variable reference has been  successfully parsed
  FLOAT_T value = 0;
  Var_ref->GeValue(value);
  /// now holds the value of variable matrix1, at position [3][2]
}

cVarRef  * var_ref = context->VarRef("matrix1[2][3]");
if (var_ref != NULL) {
  /// variable reference has been  successfully parsed
  var_ref->SetValue(10.5);
  /// now the value of variable matrix1 at position [3][2] is set to 10.5
}
```

For the first look it seems, that the variable reference concept only saves typing work for the application programmer in the sense that structure information are incorporated into the construction of the used object and therefore must not be specified when calling Set or Get.

This is true but the usage of variable references instead of variables has some more advantageous implications:

1. Variable reference structure can be supplied dynamically using other variable references instead of constant indices
2. A variable reference can act as target of data change notifications. This offers the possibility to use variable references as end points for CCS data change events. A widget, that visualizes a variable value, uses this behaviour to automatically change visual value representations in the GUI.

7.1Unlimited Variable References

To explain the usage of the first case let us consider a variable reference as a tree that denotes the value of a variable at a specific position within the variable structure. For this purpose we extend our demo database and add two additional variables called

index1 and index2. These variables are standard variables of integral type that can act as indices into the two – dimensional structure of matrix1.

```
database demo device=device1 {

   matrix1[5][10] float(2)
    vartype=set
    description="varioable definition for demonstration purpose"
    text="matrix1"
    dim1text="index 1"
    dim2text="index 2"
    persist=file;

  index1 long
    description="variable index helper variable "
    text="index1";

  index2 long
    description="variable index helper variable "
    text="index2";

};
```

Using the index variables it is possible to use a variable reference of the following form:

```
Matrix1[index1][index2]
```

When getting or setting the value of this variable reference the selected position of the affected value depends on the value of the variable index1 and index2. The next code snippet shows the usage of this scenario.

```
cVariable * i1 = context->Variable("index1");
if (i1 != NULL) {
  i1->Set(2, -1, -1, -1, -1);
}
cVariable * i2 = context->Variable("index2");
if (i2 != NULL) {
  i2->Set(3, -1, -1, -1, -1);
}
cVarRef  * var_ref = context->VarRef("matrix1[index1][index2]");
if (var_ref != NULL) {
  /// variable reference has been  successfully parsed
  FLOAT_T value = 0;
  Var_ref->GeValue(value);
  /// now holds the value of variable matrix1, at position [3][2]
}
```

Instead of using variable references ist is possible to use free classed user functions to determine index values of variable references.

```
Matrix1[fact(index2)][index2]
```

This expression is also a valid variable reference where the index into the first dimension of the variable is determined by a function call to the factorial function fact(v) with the value of index2 supplied as argument.

Fig. 7-1 shows the resolution tree for this simple variable reference.

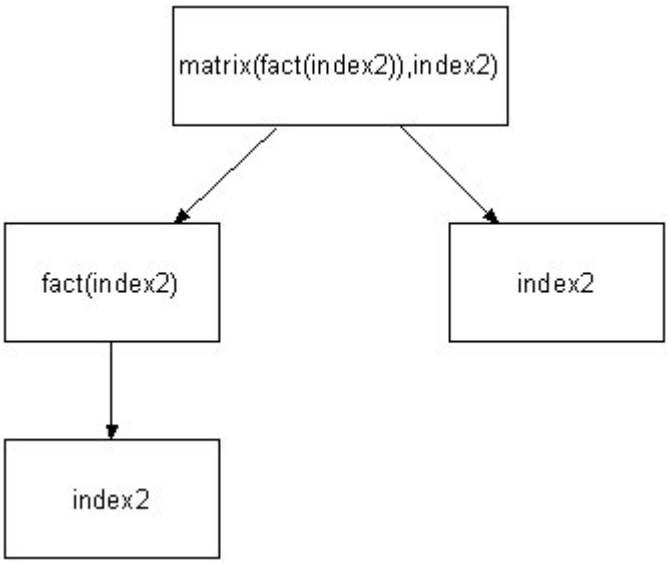


Fig. 7-1 Resolution tree for a simple variable reference

A variable reference is unlimited in the sense that the resolution tree is not limited neither in depth nor in width.

With this explanations in mind the integration of CCS data into the Qt GUI Framework should be more comprehensive.

7.2Variable references as notification end points

To explain the usage of the second case let us consider a variable reference that shows in any way the value of a variable reference. If the application programmer wants to keep track with the current value the application programmer must implement a mechanism to poll the variable for example every 100 milliseconds.

If there were an automatic event that signals the variable reference whenever the associated value has changed polling the value actively would not be used to keep track with the value of the variable reference. The C++ object cVarRef ahs a method to register with the CCS database for data change events. When a variable is registered it receives data change events when the selected value of the variable has changed.

The received cDataChange object used to signal this situation holds the new value and some context information about the variable.

Bound to a visual software component like a console, control or widget the variable reference can change the visual representation when getting a data change notification.

8Qt GUI Framework

In general the type of user interface used in the Shacira Framework is neither restricted to a specific user interface technology nor is it restricted to a specific programming language.

Actually a user interface based on the Trolltech Qt library has been developed to support comfortable designing of user interfaces with a full interface to CCS Services. The rest of references this Qt based GUI Framework as the GUI Framework.

A GUI framework should use both the synchronous and the asynchronous interface to CCS Services to fully benefit from the functionality offered by a CCS Service. The Qt GUI Framework uses all functionality that is offered by a CCS Service.

8.1Information flow in a Shacira application

Figure 8-1 shows the information flow within Shacira applications. Dotted arrows denote asynchronous uncoupled communication via CCS-Events. Integral arrows specify synchronous communication via a Shacira context.

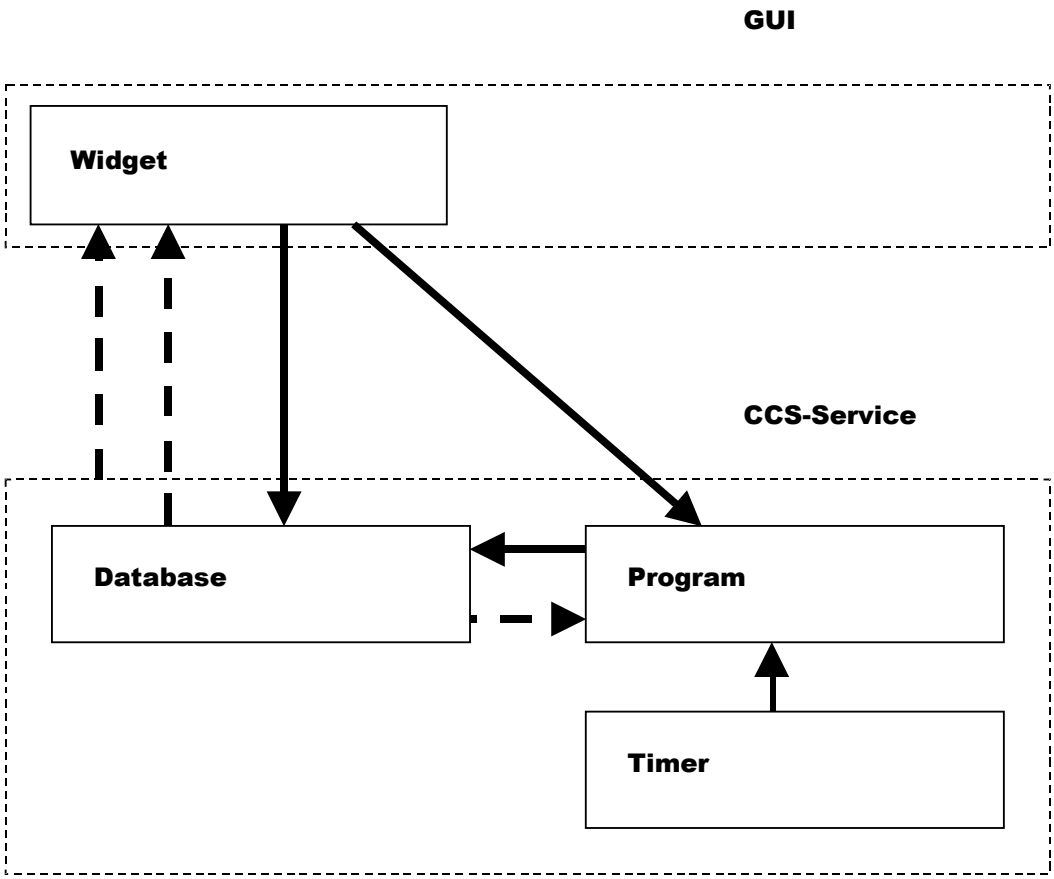


Fig. 8-1 Information flow between CCS service and GUI application

8.2Software architecture of the GUI Framework integration

Together with CCS services Shacira comes with a GUI-Framework that is based on Trolltechs Qt graphics library. The Shacira GUI-Framework consists of a frame class that can hold an arbitrary number of so called information pages. The application frame structures these information pages and offers a mechanism to navigate through the pages.

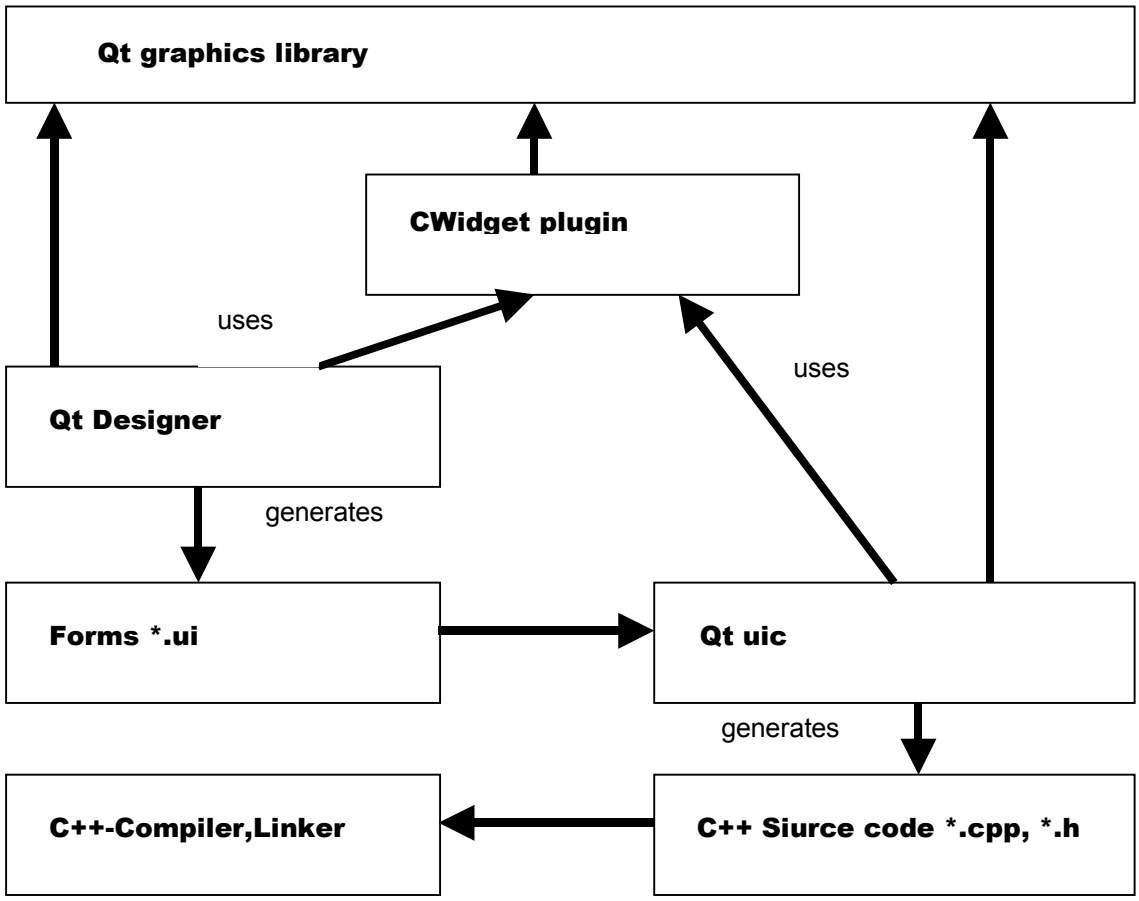


Fig. 8-2 Software Production Environment Integration of the GUI-Framework

8.3User interface structure

The structure of a GUI application in the Shacira Framework consists of an application frame that houses a couple of page groups where every page group houses a couple of information pages.

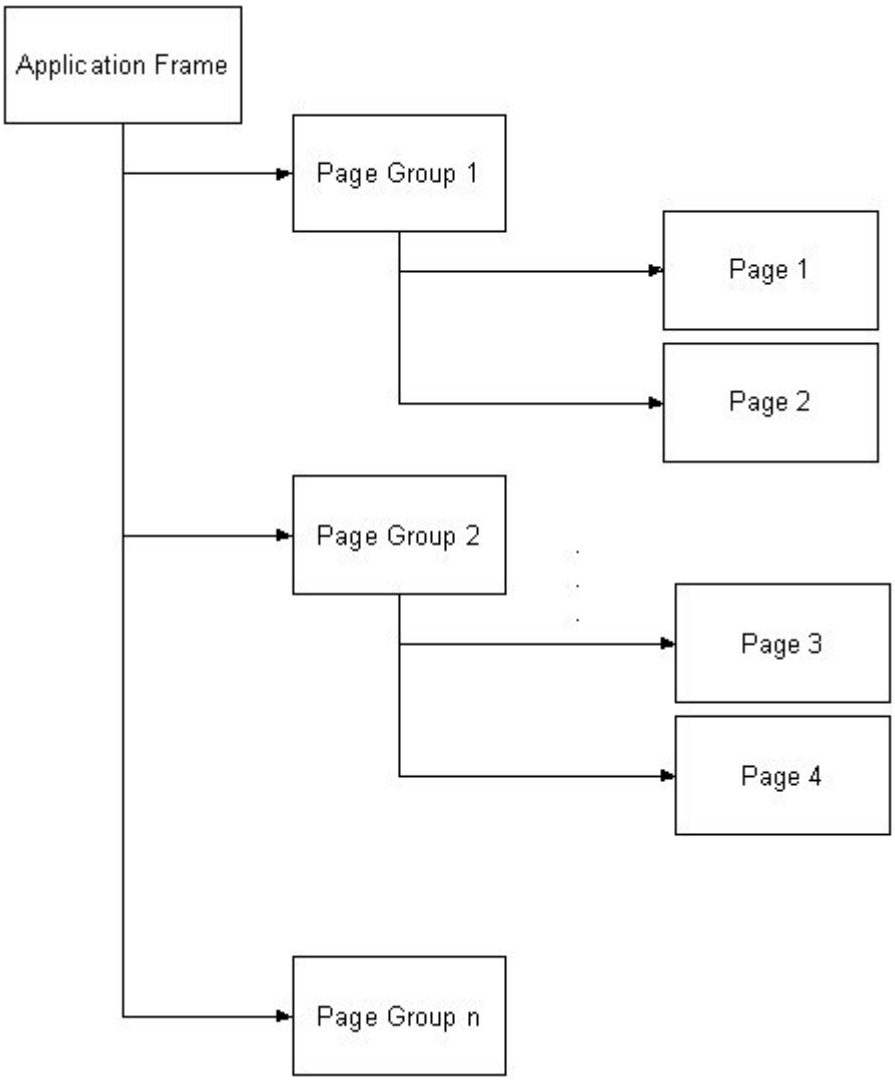


Fig. 8-3 General structure of a user interface

An Application frame is a Qt Form that acts as frame for all user defined pages of an application. CAppFrame is the base class for an application frame and it is realised as Qt widget.

The application frame structures and organises information pages that are the top level widgets to integrate different widgets for showing and manipulating CCS data.

To design and layout information pages Trolltechs Qt Designer can be used. Qt Designer enables application programmers to layout and design information pages in a Wysiwyg environment.

To interface to CCS services from within information pages, a couple of so called CCS aware Widgets (CWidgets) have been developed on top of the Qt plugin mechanism. The Qt plugin mechanism is a possibility to realize UI-Controls (widgets) that have builtin custom specific functionality. CWidgets integrate fully with Qt Designer. Every widget can be placed anywhere on an information page. The functionality of every CWidget can be configured to specific needs with the property Editor of Qt Designer.

The major mechanism to integrate a widget with CCS data access are variable references that are incorporated naturally into the widget definitions, and that are part of almost every CWidget.

8.4General CWidget functionality

To interface with a CCS service every CWidget has virtual methods to connect to a CCS service:

```
virtual void CCSSetContext(NODE_PTR node, CONTEXT_PTR context);
virtual void CCSNewValue(CONST_STRING_T value, ULONG_T id, ULONG_T time_offset,
                         UCHAR_T data_type, ULONG_T size);
virtual void CCSNewValue(BUF_T buf, ULONG_T id, ULONG_T time_offset,
                         UCHAR_T data_type, ULONG_T size);
virtual void CCSEvent(TRANSIENT_OBJECT_PTR object);
```

CCSSetContext sets the widget context when the system is starting up. The first parameter node has currently no meaning, the second parameter context is a pointer to the CCS context, the widget can connect with.

CCSNewValue is called every time a variable reference, that is associated with the widget changes his value. The argument id identifies the variable reference that caused the data change event. If a widget uses more than one variable reference different variable references can be distinguished by using a unique id when initializing the reference in CCSSetContext. When a CCSNewValue is called on the widget, the id helps to identify the source of the data change event.

The argument time_offset carries the time difference to the last occurrence of a data change event of an associated variable reference.

The argument data type supplies the data type code of the associated variable.

The argument size contains the element size of the (value size) of the associated variable.

The method CCSNewValue comes in two versions. The first version supplies a string type value, the second version supplies a BUF_T (memory pointer) value. The second version is used for variables of type SH_BYTE.

Fig. 8-4 shows the event propagation originating elsewhere in the associated CCS service up to a widget within the user interface.

The framework dispatcher is a mechanism that selects and routes events originating in the CCS service to the CWidgets that are linked to specific events.

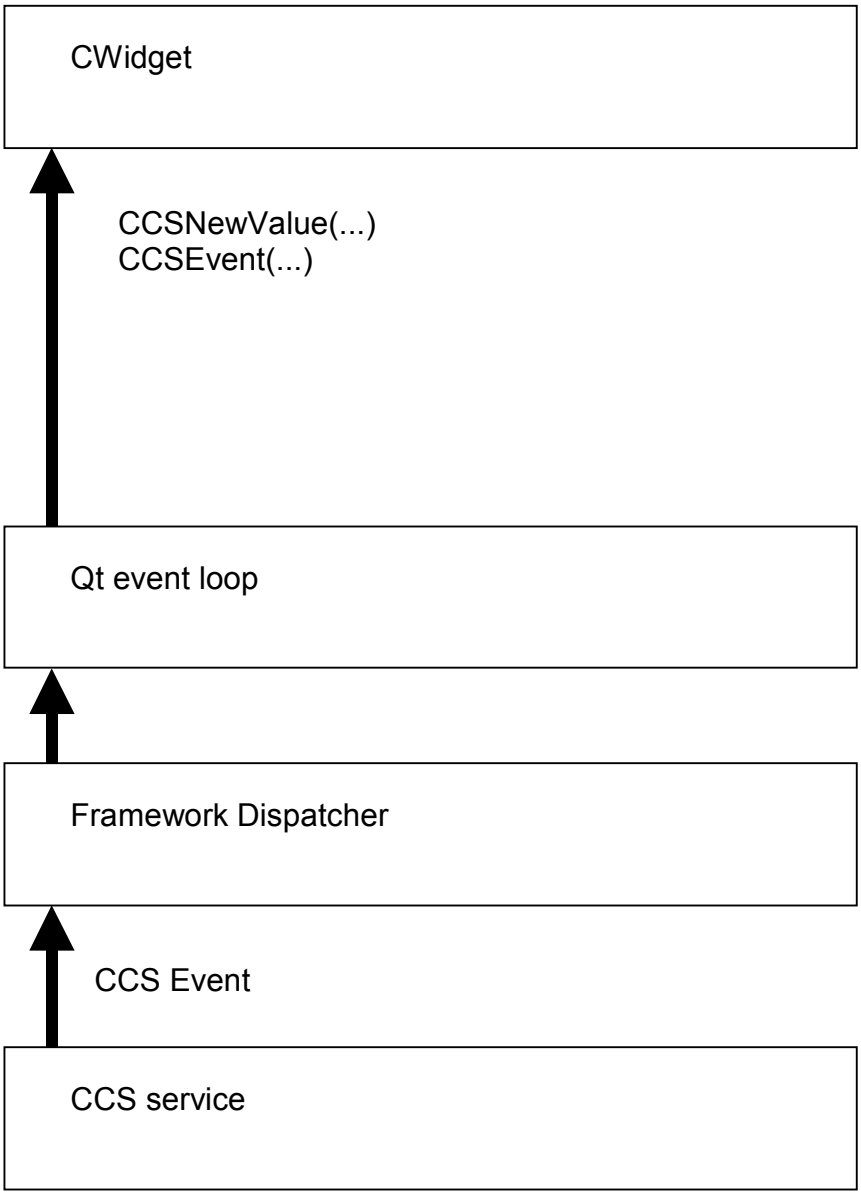


Fig. 8-4 Event Propagation from Services to a CWidget

From an abstract point of view every CWidget interfaces to a CCS service through one or more variable references. If variable values change within the CCS service the widget is signalled and receives the new value of the associated variable reference.

As a consequence the widget changes its internal state. The new internal state of the widget will be expressed in the visible area with the next repaint of the widget.

All CWidgets that can show CCS data items behave in this way. For the visualization of data there is no direct connection to a CCS service. A CWidget is an autonomous GUI-Control that reacts on CCS service events.

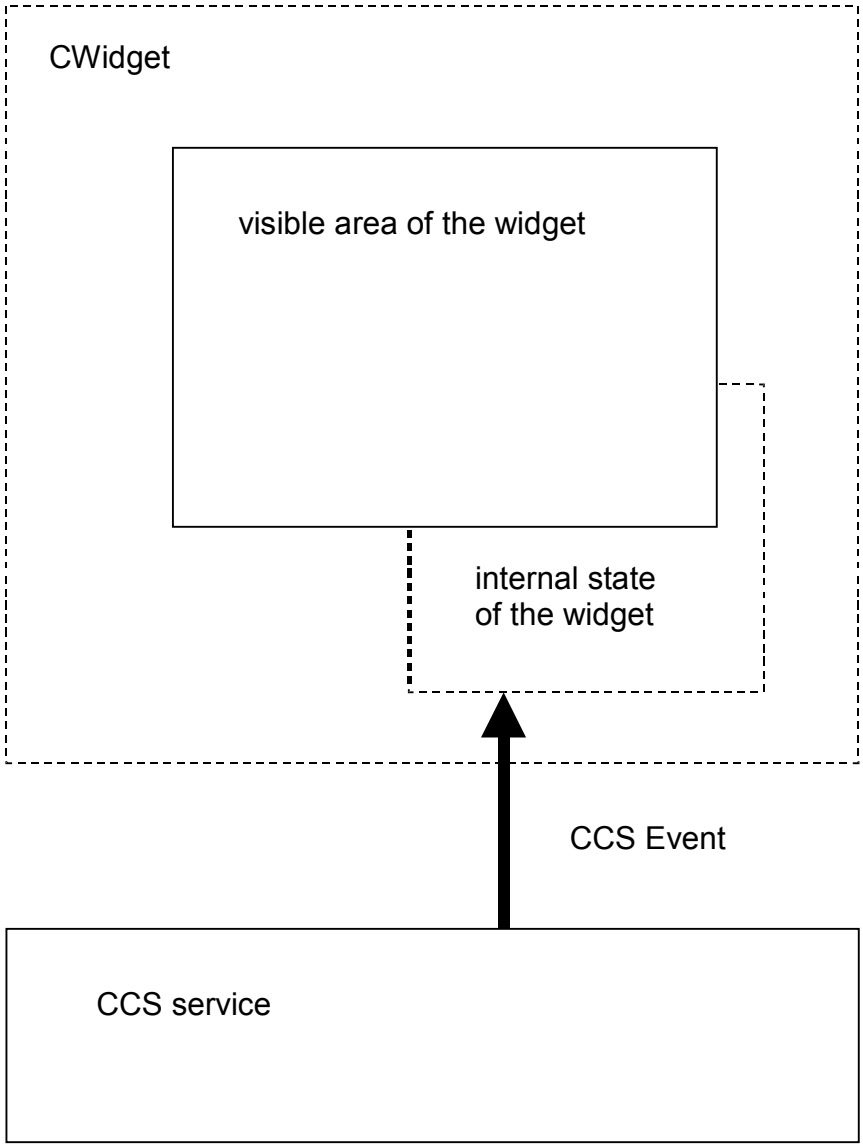


Fig. 8-5 General CWidget update functionality

When using widgets that allow data entry this situation is extended to synchronous communications between a widget and a CCS service. In this case the variable references are used to receive data change events **and** to manipulate CCS data.

8.5CWidget Data Input functionality

Data input of a CWidget is - like CWidget update functionality - based on a variable reference. Data input is realized as a three step commit mechanism.

1. Call a plausibilty function on the input.

2. Call a user function on the input.
3. Set the associated variable reference value to data input.

If one of the three steps fail, the subsequent steps are not executed and the input value will be rejected. This input mechanism is used by every CWidget that offers data entry functionality.

8.6General CWidget properties

For different types of information representation and manipulation, the CWidget plugin offers different widget types. There is a set of properties and functionality that every CWidget will expose.

Dark (blanking) functionality

Every CWidget is involved into the dark (blanking) functionality of the GUI framework. The blanking functionality allows application programmers to bind every widget to a C-style function called dark function. This function determines via return value the visible state of a widget. With this mechanism every widget can be shown, hidden or disabled on base of a free programmable function.

As a consequence every CWidget offers the property DarkFuncRef

Property **DarkFuncRef:** This property of type string holds the function reference that links to a user defined function to decide if the corresponding widget should be shown, disabled or hidden.

Event receiver

Every widget that exposes a variable reference receives data change events. All other transient objects can be received by configuring the EventFilter property. All other transient objects that originate from CCS services are called general events.

Every widget can act as receiver of general events originating in the CCS service. A widget can filter the type of events it wants to receive and act upon. By default a widget does not receive any general events.

Property **EventFilter:** This is a property of type enumeration that enables to configure a widget to receive non standard events from a CCS service (data change events are standard events). Every CWidget has this property.

Framework specific signal slot mechanism

To overcome the need to directly connect signals and slots of two widgets, a similar mechanism based on user functions has been added to the GUI framework.

A widget can carry a so called qualified name and act as signal sender. All widgets that want to be signalled by this widget can register with the sender. If a widget wants to listen to a widget that emits signals, the property listen must be set to true and the property ListeningTo must be set to the qualified name of the signalling widget.

The emitted signal is an integer code that identifies signal origin.

To make this mechanism freely programmable, 2 user functions can be supplied to adapt the necessary functionality. Normally every widget signals exactly one signal in a certain situation (for example SIG_BUTTON_CLICKED when clicking a button). The signal filter function changes the signal code to a user specific signal code. After a signal was emitted, all slot functions of the widgets, that listen to the signalling widget are called. The user specific (filtered) signal is supplied to the slot function.

This mechanism is similar to the Qt signal slot mechanism but it is more general and easier to use. The communication mechanism is restricted to integer coding.

Property **QualifiedName :** This is a property of type string that supplies a qualified name to the widget. This is a name that can be referred to when listening to signals emitted by other widgets in an application. This name does not change when a widget is cut and pasted to another location within the GUI.

Property **Listening :** This is a property that sets a widget to the listening state, that means a widget listens to signals emitted by other widgets in the application.

Property **ListeningTo :** This property lets a widget listen to the signals emitted by another widget whose qualified name matches this property.

Property **GUISignalFilterFunc :** This is a property of type string that links to a user defined function that filters signals emitted by the widget. Every widget emits signals of type unsigned long. Other widgets within the GUI can listen to this signals. Before emitting a signal the associated filter function substitutes the original signal with a user defined signal.

Property **GUISlotFunc :** This is a property of type string that links to a user defined function that is called when a widget receives a signal from another widget.

Widget Access Properties

Besides the dark functionality every CWidget has properties to control widget access based on a user group id, that can be set using the class cUserInfo and some methods of the CAppFrame API.

The properties describe widget access regarding the current user group id. 8 user groups can be set and controlled by every widget. The properties are named Group0 to Group7. For every group access property following values can be specified:

1. **WriteAccess** : The user has full unrestricted access to the widget. Input is enabled.
2. **ReadOnlyAccess** : The user has read access to the widget. Input is disabled.
3. **NoAccess** : The has no access to the widget. The widget is hidden for the user.

When a widget carries a dark function there is a potentially a conflict between the return value of the dark function and the current access specification for the specific user. In this case the following rules apply.

dark result	widget access	widget state
ElementFree	WriteAccess	enabled
ElementFree	ReadOnlyAccess	disabled
ElementFree	NoAccess	hidden

elementDisabled	WriteAccess	disabled
elementDisabled	ReadOnlyAccess	disabled
elementDisabled	NoAccess	hidden
elementHidden	WriteAccess	hidden
elementHidden	ReadOnlyAccess	hidden
elementHidden	NoAccess	hidden

Variable References

Every CWidget can be bound to a CCS variable value by specifying a variable reference. The variable reference then receives data change events (transient objects of type cDataChange). A data change event signals that the widget should show (communicate) this change.

Property **VarRef:** This property of type string holds the variable reference.

Not every widget offers this property because in the case of general containers like CFrame no semantics is defined for this situation.

8.7Input Widgets

Common to all input widgets is a triple of properties. PlausFuncRef, a function ereference that implements input validation, UserFuncRef a function reference that implements subordinate processing and VarRef, a variable reference as the target for input operations.

Property **PlausFuncRef:** This is a property of type string that links to a user defined function that decides if the input value should be rejected or not. The action to take is signalled via return value. This property will only be used with widgets that enable data input.

Property **UserFuncRef:** This is a property of type string that links to a user defined function that decides if processing of input data should be continued or cancelled. The action is signalled via return value. This property will only be used with widgets that enable data input.

Property **VarRef:** This is a property of type string that links to an element value of a variable defined in the underlying data model. In general every CWidget that shows or manipulates variable data has this property.

8.8Application Frame CAppFrame

An Application frame is a Qt Form that acts as frame for all user defined pages of an application. CAppFrame is the base class for an application frame and it is realized as Qt widget.

The application frame adds special functionality that is targets application specific issues like language switching.

Application frames are layed out using Qt designer. The concrete design of the application frame is not restricted. But to profit from the builtin functionality there are some aspects that must be considered. An application specific frame must be derived from the base class CAppFrame.

virtual methods of an application frame object

CAppFrame contains some virtual methods that must be reimplemented by the concrete frame class. The derived application frame overloads these methods to adapt to the internal mechanisms of CAppFrame like navigation facilities resource management and builtin functions.

*virtual CFrame * GetNodeContainer();*
This method supplies a pointer to a CFrame type container where node navigation buttons are to be placed.

*virtual CFrame * GetGroupContainer();*
This method supplies a pointer to a CFrame type container where the page group navigation buttons are to be placed.

*virtual CFrame * GetPageContainer();*
This method supplies a pointer to a CFrame type container where the concrete information pages are to be placed.

*virtual CFrame * GetSoftkeyContainer();*
This method supplies a pointer to a CFrame type container where the softkey buttons of the pages are to be placed.

*virtual CFixText * GetPageHeader();*
This method supplies a pointer to a CFixText type widget where the title of an individual information page will be placed.

*virtual CFixText * GetStatusBar();*
This method supplies a pointer to a CFixText type widget where status information is to be placed.

virtual void LoggedOff(CONTEXT_PTR context, USER_INFO_PTR_T user_info,
BOOL_T automatic);
This method is called, when a logoff occurs. The circumstances of the logoff is communicated via the argument automatic. The argument context is a pointer to the current context that can be used to access or set variable information. The argument user_info is a pointer to a copy of the user info at the time when the logged off user has lagged in.

Global functions to bootstrap the user interface

To hook an application frame and the corresponding page list into the runtime system, the application specific code must offer two global functions.

*CAppFrame * CreateAppFrame();*

CreateAppFrame creates the application specific frame and returns ist as a pointer.

*PageList * CreatePageList(QWidget * parent, cNode * node)*

PageList creates the list of pages that must be managed by the application frame and returns this list.

The overloaded methods of CAppFrame and these two functions will hook all custom specific GUI-code into the base framework.

CAppFrame methods

An application frame object supplies the application programmer with a lot of functionality. A programmer has access to this functionality in every GUI function.

Page Stack methods

void ShowPage(QCString page_name);
Shows the page with the name page_name. If necessary the related page group will be switched.

void ShowLastPage();
Shows the previous page selected. If necessary the related page group will be switched.

void Refresh(int refresh_type, BOOL_T delayed = false);
Executes a refresh of type refresh_type on the user interface. When delayed is true the refresh operation is delayed until the calling function has been executed.

The following refresh_types are possible:

- **EVENT_REFRESH**: all dark functions are executed immediately to adapt the user interface appearance to a new state.
- **DATA_REFRESH**: all values of all variable references within the user interface are read and fed into the standard event signalling mechanism. This function behaves as if all values of the variable references will change.

*CPage * ActPage();*
Returns a pointer to the page that is currently shown (visible).

*CPage * GetPage(CONST_STRING_T page_name);*
Returns a pointer to the page specified by page_name. Returns NULL if this page is not present.

QCString ActPageName();
Returns the name of the page that is currently shown (visible).

Language switching related methods

bool SetNewLanguage(CONST_STRING_T language);
Switches the language to the language specified by the argument language. This operation succeeds, when a translation file with name <language>.qm exists.

QCString ActLanguage();
Returns the current language as string.

Printing related methods

void PrintActPage(ULONG_T flags = 0);

Prints the current page to the system printer.

void PrintPage(CONST_STRING_T page_name, ULONG_T flags = 0);
Prints the page specified by page_name to the system printer.

QImage GetImage(CONST_STRING_T page_name = NULL, ULONG_T flags = 0);
Returns an image of the page specified by page_name instead of printing it on the system printer.

The flags argument can be supplied to control print output:

PRINT_FRAME prints the page together with the application frame.

Dialog and wizard related methods

*void RegisterDialog(QDialog * dialog);*
Registers a dialog with the app frame.

*void RegisterDialog(QWizard * dialog);*
Registers a wizard with the app frame.

int ExecDialog(CONST_STRING_T dialog_name);
Executes a dialog (wizard or dialog).

*QWizard * Wizard(CONST_STRING_T name);*
Returns the wizard object with name name, if such a wizard has been registered, NULL otherwise.

*QDialog * Dialog(CONST_STRING_T name);*
Returns the dialog object with name name, if such a dialog has been registered, NULL otherwise.

User management methods

void SetGroupId(ULONG_T id);
Sets the group id of the current user.

void SetUserInfo(USER_INFO_PTR_T user_info);
Sets the whole user information.

USER_INFO_PTR_T GetUserInfo();
Returns a pointer to the user information of the current user.

void Login(USER_INFO_PTR_T user_info);
Logs in a new user. The whole user information is supplied as argument

void Logoff(BOOL_T automatic = false);
Logs off the current user. The argument automatic signals an automatic logoff (done by the system).

Help related methods

void ShowHelp();
Executes (pops up) the Help dialog.

void HideHelp();
Closes (hides) the Help dialog.

BOOL_T HelpActive();

Returns true if the help dialog is active false otherwise.

8.9CWidget types

Container widgets

CFrame

This widget is derived from the Qt QFrame widget.

CGroupBox

This widget is derived from the Qt QGroupBox widget.

CbuttonGroup

This widget is derived from the Qt QButtonGroup widget.

Button widgets

CWidget Button widgets are derived from the associated Qt button widgets and extended by the general CWidget functionality.

CtoolButton

This widget is derived from the Qt QToolButton widget.

CPushlButton

This widget is derived from the Qt QPushButton widget.

Data input widgets

CStateButton

This CStateButton is a special button widget that can be supplied with a variable reference. The value of the associated variable reference is shown by distinct icons or textual representation. At most 5 distinct states can be shown.

Clicking a CStateButton changes the value of the associated variable reference to the next valid state.

The mapping from states to variable values can be configured through properties within the property editor of Qt Designer.

CReqValue

A widget of type CReqValue can be used to show and manipulate textual representations of variable data.

CIndReqValue

A widget of type CIndReqValue can be used to show and manipulate textual representations of variable data, that are indexed.

CRadioButton

A widget of type CCheckBox can be used to show and manipulate 0/1 representations of variable data.

CCheckBox

A widget of type CRadioButton can be used to show and manipulate 0/1 representations of variable data.

CComboBox

A widget of type CComboBox can be used to show and discrete integer representations of variable data.

Display widgets

CFixText

CActValue

CGraphic

CProcGraphic

CListView

CVideo

CAlarmWidget

CTable

8.10Organization of custom specific GUI code

To organize custom specific GUI code the following directory structure is recommended:

Starting from a custom specific root directory <customdir> the following subdirectories are used to organize GUI-descriptions and functions.

Common-Directory <customdir>/Common (<commondir>)

The common-directory should contain all code that is common to the development of all applications. A common-directory is structured into subdirectories to contain differ-

ent types of custom specific code. The structure of the common-directory acts as template for subdirectories containing code for application variants. A common may contain subdirectories with the same structure containing the code for an application variant.

Subdirectory **Widgets** <widgetdir> contains custom specific Qt widgets that inherit from existing widgets like Qt-widgets or CCS-aware CWidgets.

Subdirectory **Forms** <formdir> contains page descriptions (.ui) designed with Qt designer and the generated files of the tools uic and moc.

Subdirectory **Funcs** <funcdir> contains a data model, user function descriptions, the generated files of mdlc and the function implementations.

Subdirectory **Programs** <programsdir> contains the code for CCS-service control programs.

Subdirectory **Init** <initdir> contains the code for CCS-service initialisation functions.

Subdirectory **Text** <textdir> constant application specific text, that is not part of the GUI (that is not located somewhere in widgets or pages).

Custom-Base <customdir>/CustomBase

The custom base directory contains the three base files that act as object and function factories for custom specific devices, custom specific control programs and user functions.

9How to Implement GUI functions

Custom specific hook functions are used as CCS extensions referred to in the data model of a CCS service or as GUI extensions that must be declared in a declaration file.

A user function must be associated with a function class. The function class preceeds the function declaration and implies the semantic and circumstances of function execution.

9.1Functions that extend the GUI

Dark functions

Dark (blanking) functions to show, hide or restrict GUI widgets. The return type of a dark function is always a long value. Return values can not be specified for plausibility functions.

The return value indicates the action to be taken.

elementVisible: show the widget

elementDisabled: show the widget but restrict (disable) input

elementHidden: hide the widget

Plausibility functions

Plausibility functions are used to check the validity of data input. Data input is supplied as first argument to the function. This is an automatic argument and must not be declared within the function declaration. The return type of a plausibility function is always a long value. Return values can not be specified for plausibility functions.

The return value indicates the action to be taken.

actionProceed: input is valid and proceed subsequent processing

actionReject: reject input and terminate subsequent processing

actionIgnore: accept input but terminate subsequent processing

User functions

User functions are used to activate an action after the successful validity check on data input. Data input is supplied as first argument to the function. This is an automatic argument and must not be declared within the function declaration. The return type of a plausibility function is always a long value. Return values can not be specified for plausibility functions.

The return value indicates the action to be taken.

actionProceed: accept input and proceed subsequent processing

actionReject: reject input and terminate subsequent processing

actionIgnore: accept input but terminate subsequent processing

Button functions

A button function is similar to a user function without automatically supplying input data in the first argument (there is no input data associated with buttons).

The function prototype generated by mdlc differs from the original declaration. In general one or more parameters are added in front of the parameter list. Which of the so called auto parameters are generated depends on the function class.

Function class	Parameter	Parameter type	semantics
Dark (blanking)	_context	cContext *	Pointer to CCS context
	_*abstract*_widget	WIDGET_PTR	Pointer to abstract widget
Button	_context	cContext *	Pointer to CCS context
	_*abstract*_widget	WIDGET_PTR	Pointer to abstract widget
Plausibility	_context	cContext *	Pointer to CCS context
	_*abstract*_widget	WIDGET_PTR	Pointer to abstract widget
	_input	const char *	Supplied input value
User	_context	cContext *	Pointer to CCS context
	_*abstract*_widget	WIDGET_PTR	Pointer to abstract widget
	_input	const char *	Supplied input value
GUISignalFilter	_context	cContext *	Pointer to CCS context
	_*abstract*_widget	WIDGET_PTR	Pointer to abstract widget
	_signal	unsigned long	Original widget signal
GUISlotFunc	_context	cContext *	Pointer to CCS context
	_*abstract*_widget	WIDGET_PTR	Pointer to abstract

			widget
	_signal	unsigned long	Original widget signal
	_sender	WIDGET_PTR	Pointer to sender of the signal
Embedded	_context	cContext *	Pointer to CCS context
Free	-	-	-

Tab. 7-1 Auto parameters of GUI function classes

The declared input parameters of a user function follows the set of auto parameters. The first of the auto parameters is a pointer to the CCS context. The CCS context pointer can be used to accesss whatever CCS data is used to realize the function see chapter 8.

10How to Implement Application Specific Widgets

11 System Startup Procedure

When a Shacira application starts up some rules must be considered. These rules differ depending on the startup mode of the Shacira apllication (client or server).

11.1 Client Startup

11.2 Server Startup

Startup of a GUI based Shacira application starts with the user interface as master and the CCS service as Slave. Startup consists of 5 phases that can be assigned either to the GUI or the CCS service.

1. Create and display of the Startup Window (GUI).
2. Loading (instantiating and creating) the CCS service context (CCS).
3. Create information pages and link all the widgets into the CCS service (set the context, create variable and function references).
4. Starting the CCS service (load peristent data, starting devices, programs, and general infrastructure, synchronize data with devices, …).
5. Display the Graphic User Interface.

Custom specific functionality can be hooked to the Startup process by means of inplementation of 2 functions that are called by the Startup process whenever a specific step in the procedure is reached. These Functions are _MdlInit (CCS service) and _GUIInit(GUI). The specific step within the start up process is communicated the function argument step.

Figure 11-1 shows an overview of the startup procedure. The grey shaded area denotes initialisation activities that are static. This means, the states generated by the functionality of this phases persists subsequent start ups.

The static phases can be assigned to fix suspend activities.

The states of all other phases can differ from one start to the next because they take into account data that persists system shutdown.

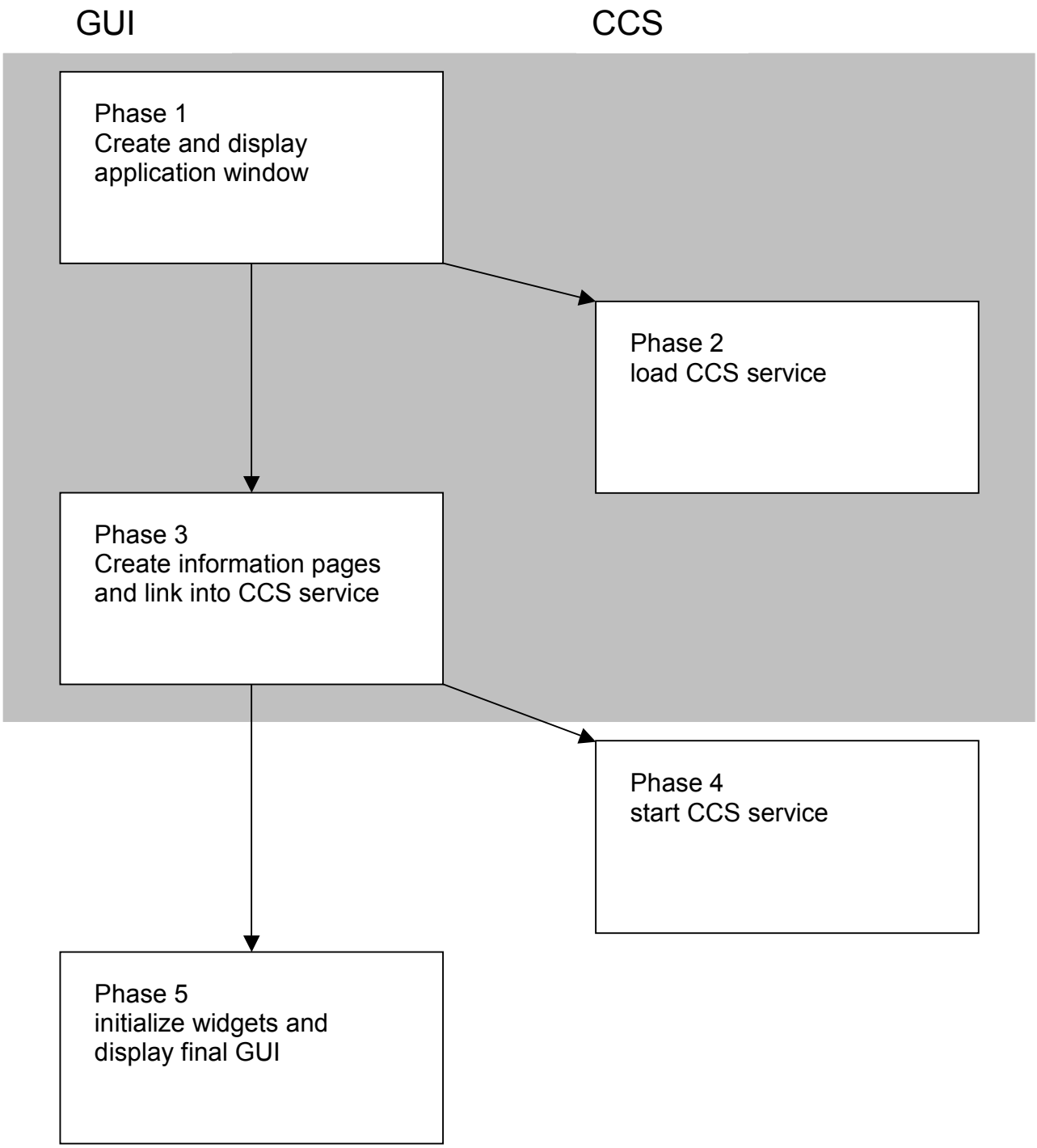


Fig. 11-1 Start Up Procedure

Initialisation steps of the CCS service

step	reached state	consequences
Phase 2		
SH_PRE_CONTEXT_INSTANTIATION	Nothing happened at all	

SH_PRE_CONTEXT_CREATION	*CCS has parsed the associated data models and will call the method Create immediately on the context*	
SH_POST_CONTEXT_CREATION	CCS created the context successfully.	Resources like memory and device mappings are already created for every variable of the context. Now it is possible to set and get data from a variable. Nevertheless variable data is not yet initialized.
SH_POST_CONTEXT_INSTANTIATION	The context object is completely instantiated.	
Phase 4		
SH_PRE_SYSTEM_STARTUP	First step of this phase.	
SH_PRE_LOAD_DATA	First dynamic step. After this step the process will check for valid persistent data and load it, or otherwise initialize all data.	
SH_INITIALIZE_DATA	This step only occurs, when the whole data is initialized (no valid persistent data found)	
SH_DATA_INITIALIZED	This step occurs after the data of all variables is initialized.	After this step correctvariable initialisation is guaranteed (either by persistent data or a general init)
SH_PRE_UPLOAD		
SH_POST_UPLOAD		
SH_PRE_DOWNLOAD		
SH_POST_DOWNLOAD		
SH_CACHE_IS_FREE	*This step is called, after all buffers are switched of.*	From this point on full device access is guaranteed.
SH_POST_SYSTEM_STARTUP	Last step of this phase.	

Initialisation steps of the GUI

The phases of GUI initialisation are mainly determined by virtual methods that are called in specific GUI start up phases.

1. CwidgetBase::CCSSetContext(…)

Called in phase 3.

Only variable references should be created and initialised. At this point the data of variables is not yet initialised.

2. CwidgetBase::Setup(…)

Called in phase 5.

Variable data related stuff can happen here.

3. UserInitfunction

Called in phase 5 but after the virtual Setup method.

The same

12Tools

12.1Developer Tools

Data-Model-Compiler mdlc

The data model compiler mdlc.exe can be used to

- Check the syntactical correctness and validity of a data model
- Generate stub code for the implementation of user functions
- Generate text file for localisation

mdlc is a command line tool with following usage:

mdlc input=<input_file> header=<prototype_file>
table=<table_file> strings=<text_file>

input_file is the file that contains the data model.

header_file is the file where mdlc places C prototype declarations of the declared user functions.

table_file is the file where mdlc places the function address table for user functions.

text_file is the file where mdlc places the localisable strings of the data model.

mdlc can be integrated into the MS Visual C++ build process.

12.2Runtime Tools

Runtime tools are offered to simplify testing and projecting.

System Console

A sytem console (program scons) enables access to running CCS services. Using a system console, all variables of a running CCS service can be shown and manipulated.

Command line arguments

- RootName=<name of the root object> (default CCSClient)

System console offers the following commands.

lc (list contexts)

Shows available contexts (CCS services).

sc <context name> (set contexts)

Connects to a context specified by <context name> .

ac (active context)

Shows the name of the active context (this is the last context that has been successfully set with sc).

lv (list variables)

Shows all available variables of the active context. Th listing includes type and structure information.

<variable reference> (variable query)

Query the value of the variable denoted by <variable reference>.

<variable reference> = <value> (variable manipulation)

Set the value of the variable denoted by <variable reference>.

lp <program file name> (load program)

Loads an SCPL program.

ep <procedure name> (execute procedure)

Executes the SCPL procedure specified by <procedure name>.

Info Console

Info console (program icons.exe) shows CCS service messages directed to a UDP port.

Command line arguments

- Port=<port> (default 9500)

Channel Manager

The channel manager (cmgr.exe) starts and stops event channels for event driven remote communication.

13Configuration

CCS services and CCS client programs can be configured using a flexible hierarchical configuration scheme. The scheme models typed objects with typed properties. This scheme is actually implemented on top of Files similar to Windows-Ini files. Objects map to chapters, object properties map to property lists within an objects chapter.

The structure of a configuration is defined by a meta definition (similar to XML dtd) held in a file called definition file (in general the file name is Shacira.def)

The starting point of a configuration is a root object that must be specified, when starting a CCS service or one of the client programs. The type of the root object can be freely chosen for a specific configuration. In genera the type of the root object differs for different kinds of CCS application. A CCS service needs a root object of type Process, a CCS GUI application needs a root object of type UserInterFace.

Basic object property types:

Integer (integer numeric valuel)

Boolean (true, false, ja, nein)

Object Reference (a reference to another object)

String (arbitrary character sequence)

Enum (value enumeration)

Sorted lists (vectors) of the basic types.

Every configuration object has a name. The object name is the chapter name of an object description as realized actually.

Every property of an object is represented by a pair <property name> = <property value>. Property names are checked for validity (only property names specified in the configuration scheme for this object are valid), property values are checked for type correctness.

Not every property that is specified in the configuration scheme must be present in a concrete object description.

Mandatory properties are determined by the application that is to be configured.

Configuration objects of a CCS service

The following tables list the description of actually available configuration objects of Al CCS service. The Shacira.def file defines the configurable objects and their properties. It is comparable to a document type definition used in XML definitions. To get more information about user defined types refer to:

Shacira.def

Configuration object

Configurationj		Description of a Shacira configuration
Path	Vector<String>	Additional paths where configuration files

		are located
Files	Vector<String>	Names of additional configuration files
DefFile	String	Name of the file that supplies object descriptions

Process object

Process	Type	Description of a Shacira process object
Resources	obsolete	
IdleTime	obsolete	
PulseInterval	Unsigned	Interval to pulse connection information of its cells in ms
ShutDownControlTime	Unsigned	Maximuml time to shut down a shacira process
ProxyReceiver	Vector<ProxyReceiver>	List of proxy receiver objects to be instantiated by the process
ProxySender	Vector<ProxySender>	List of proxy sender objects to be instantiated by the process
Receiver	obsolete	
Sender	obsolete	
Description	String	Textual description of the shacira process
Cells	Vector<Cell>	List of Cells to be instantiated and started by the process)

Cell object

Cell	Type	Description of a Shacira cell object
CellName	String	The name of the cell. Cell names are inherited by contexts. Services are exposed via a cell name. Connections to contexts can be established using the cell name. When CellName specifies a variable name in the application data model, the content of this variable is used as cell name.
Description	String	Textual description of the shacira cell

CycleSpec	obsolete	
Devices	Vector<Device>	List of devices that should be instantiated and started by the cell
Interfaces	Vector<Device>	List of device interfaces that should be instantiated and started by the cell
Devices	Vector<Device>	List of device objects that should be instantiated and started by the cell
Programs	Vector<Program>	List of program objects that should be instantiated and started by the cell
Channels	Vector<Channel>	List of channels that should be instantiated and started by the cell
Connections	Vector<Connection>	List of connections that should be established by the cell
CorbaService	bool	If true a corba object is exposed
Active	bool	If false the cell is idle
Context	Context	Asociated cell context object. The context is parsed and created when the cell is instantiated

Context object

Context		Description of a Shacira context object
ContextName	String	Name of the context
Files	Vector<String>	List of model files that describe the context database
SymbolFiles	Vector<String>	List of symbol files used within the context

Device object

Devicej		Description of a Shacira device object
DeviceType	String	Type of device
Verbose	bool	Protocol on/off
TimingProtocol	bool	Enable timing protocol
SerialChannel	SerialChannel	Reference to a serial channel object
BaudRate	BaudRate	Device baud rate if needed
Parity	Parity	Device parity if needed
Handshake	Handshake	Device handshake if needed

Charlen	Charlen	Device character size if needed
StopBits	StopBits	Device stop bits if needed
StartBits	StartBits	Device start bits if needed
ReadPort	Unsigned	UDP read port for PPCCOM over ethernet
WritePort	Unsigned	UDP write port for PPCCOM over ethernet
Host	String	Server host name or IP-Address (server is PLC) for PPCCOM over ethernet
IscosNo	Unsigned	Bus address of PLC
BaseDevice	Device	Base device for cascading device structures

Program object

Program		Description of a Shacira program object
ProgramType	String	
DebugLevel	Unsigned	Switch for debug trace
IdleTime	Unsigned	Cycle interval in ms

Serial Channel object

SerialChannel		Description of a Shacira serial channel
ChannelType	SerialChannelType	Type of the serial channel CHANNEL_STANDARD Standard com port CHANNEL_SOCKET Socket based serial channel W&T spec CHANNEL_VSOCKET Socket based serial channel 2i spec
Verbose	bool	Protocol on/off
PortName	String	Name of the com port (only interpreted by a standard channel)
IPAddress	String	IP-Address of the service (only interpreted by the socket channels)
RXPort	Unsigned	UDP-read-port (only interpreted by the socket channels)
TXPort	Unsigned	UDP-write-port (only interpreted by the socket channels)

IsBus	bool	Set to true when a bus based protocol is used (only for RS 485 or RS 422 possible)

Connection object

Connection	Type	Description of a Shacira connection object
From	String	Name of a channel or program as source of the connection
To	String	Name of a channel or program as destination of the connection

Channel object

Channel	Type	Description of a Shacira channel object
ChannelType	ChannelType	Type of the channel object LocalChannel is a channel that does not cross process boundaries RemoteBridge is a channel that crosses process and network boundaries
To	String	Name of a channel or program as destination of the connection
RemoteBridge	bool	Also local channels are enabled to act as remote bridge if RemoteBridge is set to true
RemoteName	String	Name of the event channel to be used for remoting (only interpreted when RemoteBridge is true)

14Shacira API

The Shacira API can be from different points view:

- The view of an application programmer
- The view of a system programmer

14.1Application Programmer View

cResources class

The cResources class adds static functionality to a Shacira process. It serves as carrier of global informations that can be exchanged among the different components even between the GUI interface code and the CCS service code.

cResources manages globally shared resources such as path specifications for the location of different system files.

It offers functionaliity for logging and tracing that can be used elsewhere in the application code.

cContext class

The most interesting object that can be used by an application programmer is a context object or context reference. A context object is of type cContext. A context object is directly supplied to

- All User-Functions except free user functions.
- All control programs

As a consequence application programmers are not burdened creating or getting the right context object.

A context object offers the following services:

- Create variable reference to access variable data
- Get Variable objects to access variable data
- Create user function references to activate functionality

With context objects full access to data modelled in the data model represented by the context is possible.

cVarRef class

The cVarRef class is the C++ object

cVariable class

cUnitDef class

Static objects

Static objects are C++ objects that are instantiated when a CCS service starts up and live until system shutdown. Static objects are can raise events (transient objects) and "raise" them to propagate this objects over the CCS channel system. Static objects are the origin of CCS events.

The following classes inherit from cStaticObject:

- Programs (cProgram)
- Devices (cDevice)
- Channels (cChannel).

Transient objects

Transient objects are C++ objects that carry information in the asynchronous communication mechanism of CCS services. Transient objects have methods for serialization and constructing. This property enables the CCS channel system to send asynchronous objects over network boundaries.

Transient objects are generated by static objects and propagated through the method RaiseEvent(object).

cTransientObject class

Specific transient objects

cDataChange class

The cDataChange transient object is the most used object within CCS. A data change object communicates state changes of variable values.

cAlarm class

The cAlarm object communicates alarms evolving in arbitrary static objects of CCS services.

cInfo class

The cInfo class communicates informations over the channel system.

cJobSpec class

A cJobSpec object specifies the description of a set of variable values.

cJobEntry class

A cJobEntry object communicates a set of variable values.

Error Handling with cError class

The error handling within the Shacira framework is based on objects of type cError. Concentrating on one object type simplifies the management of language independent error messages.

Objects of type cError cross process- and even network boundaries when thrown as C++ exception.

cError exceptions are handled by the controlling thread. If the thread is a non visual thread cError exceptions are logged into the shacira error log file.

If the thread is the GUI thread cError exceptions are shown through an application specific message handler. cError informations can be localized in this situation.

14.2System Programmer View

15Code Organisation

15.1Development

The Shacira development root directory points to the directory where the development framework is installed. This directory is communicated through the environment variable SHACIRADIR.

Example:

SHACIRADIR=e:\usr\prj\Shacira

Beneath the Shacira development framework the free orb OmniORB must be installed (actually version 3.04 is used). The installation directory of OmniORB is referred to as <ORBDIR> within the subsequent text. The ORB specific development directories must be directly integrated into Microsoft Visual C++ using menu Tools->Options Tab directories.

Include files:

<ORBDIR>/include

<ORBDIR>/include/OmniORB3 (in case of omniORB-3-versions)

<ORBDIR>/include/OmniORB4 (in case of omniORB-4-versions)

<ORBDIR>/include/COS

Library files:

<ORBDIR>/lib/ x86_win32

Executable files:

<ORBDIR>/bin/ x86_win32

Binaries that are needed from the OmniORB package are:

- omniidl
  The CORBA-IDL-compiler of OmniORB

Shacira binary files are copied to the directory <SHACIRADIR>/bin/win32. The path variable of the system should contain this directory.

For comfortable debugging two entries must be added to the path environment variable under windows:

1. The shacira binary path <SHACIRADIR>\bin\win32
2. The custom specific binary path that must contain all CCS-Extension dlls. In general <CUSTOMPROJDIR>\bin\win32

In the same way the path list for executable files must be extended with the Shacira bin path <SHACIRADIR>/bin/win32 in VCPP-Options.

Binaries that are needed from the OmniORB package are:

- styx.exe
  A parser generator needed to generate interpretation code for the languages SLang and EM63.

- ctoh.exe
  A tool to generate C-Headers from generated C-Files of styx.
- pp.exe
  A precompiler to translate widget descriptions with custom specific properties into C++-header files (needs ppQt.pre for the translation process).
- mdlc.exe
  The Shacira data model compiler for translation of Shacira data models.

15.2Runtime

A runtime system consists of the following parts:

Program Directory

The program directory contains the binary files of the CCS application

- Executable files (*.exe under Windows)
- Shared library objects (*.dll under Windows)

Model Directory

Default <Program Directory>/model

The model directory contains the following files:

- Data models (*.mdl)
- Additional function declarations for GUI programs (*.dec)
- Symbol files
- SCPL program files (*.pgm)

Configuration Directory

Default <Program Directory>/cfg

The Configuration directory contains the following files:

- CCS configuration schema (Shacira.def)
- Application specific configuration extensions (application.def)
- CCS configuration files (*.cfg)
- Client configuration files (*.cfg)

Log Directory

Default <Program Directory>/Log

The Log directory contains the following files:

- Log files (*.log)

Data Directory

Default <Program Directory>/Data

The Data directory contains the following files:

- Data files to hold persistent data of an application (*.dat)

Internal Directory

Default <Program Directory>/Internal

The Internal Directory is for application specific use. Every Shacira application should use the internal directory to organize application specific data.

15.3Binary Files of a CCS-Application

All binary files of a CCS-Application are located in the <program directory>. Locating binary files in one of the system directories under Windows is neither necessary nor recommended.

Standalone CCS Service Process

The program ccs.exe is a console based program that hosts CCS services. Graphical user interfaces can be separated to access data and events of the hosted CCS services.

Graphical User Interfaces

The graphical user interfaces are in general (but this is no requirement) realized as programs with the name app.exe. Graphical user interfaces may host CCS services. On the other hand a GUI can act as standalone process everywhere in the network.

16Glossary

SHACIRA

Scalable **H**igh performant data **A**cquisition and **C**ontrol Inf**RA**structure is a C++ based development framework for applications in the manufacturing area.

CCS

Core **C**ontrol **S**ervices is a service interface and implementation that acts as base service for acquisition, control and manipulation of plant floor data.

CCS service

A CCS service is a process that hosts CCS service implementations.

CWidgets

Qt based GUI controls that easily can be integrated with CCS services.

SLANG

Shacira **Lang**uage this is the language to describe application specific data models.

SCPL

Shacira **C**ontrol **P**rogramming **L**anguage is a subset of SLANG to describe procedural extensions in a CCS service.