TaskBuilder

Tutorial and Reference Manual

MBC-00013 - Issue 03 - October 2021

www.taitradio.com

Contact Information

Tait Communications

Corporate Head Office

Tait International Limited
P.O. Box 1645
Christchurch
New Zealand

For the address and telephone number of regional offices, refer to our website: www.taitradio.com

Copyright and Trademarks

Disclaimer

There are no warranties extended or granted by this document. Tait International Limited accepts no responsibility for damage arising from use of the information contained in the document or of the equipment and software it describes. It is the responsibility of the user to ensure that use of such information, equipment and software complies with the laws, rules and regulations of the applicable jurisdictions.

Enquiries and Comments

If you have any enquiries regarding this document, or any comments, suggestions and notifications of errors, please contact your regional Tait office.

Updates of Manual and Equipment

In the interests of improving the performance, reliability or servicing of the equipment, Tait International Limited reserves the right to update the equipment or this document or both without prior notice.

Intellectual Property Rights

This product may be protected by one or more patents or designs of Tait International Limited together with their international equivalents, pending patent or design applications, and registered trade marks: NZ409837, NZ409838, NZ415277, NZ415278, NZ508806, NZ530819, NZ534475, NZ547713, NZ577009, NZ579051, NZ579364, NZ586889, NZ610563, NZ615954, NZ700387, NZ708662, NZ710766, NZ711325 , NZ726313, NZ733434, NZ593887, AU2015215962, AU339127, AU339391, AU2016259281, AU2016902579, AU2017204526, EU000915475-0001, EU000915475-0002, GB1518031.8, GB1710543.8, GB2532863, US14/834609 Div. no 1, US15/346518 Div.no 2, US15/350332, US15/387026 Div., US29/614639, US62/713910, US62/729478, US62/730107, US62/767041, US62/781642, US62/778238, US9794940 Div. no 1, US20150085799, US20160044572, US20160057051, US20170142646, US20170055267 Div. no 2, US20180006844, US640974, US640977, US698339, US702666, US7758996, US8902804, US9107231, US9504034, US9559967.

This product may also be made under license under one or more of the following patents:
- US7203207, AU2004246135, CA2527142, GB2418107, HK1082608, MY134526, US8306071
- US7339917, AU2004246136, CA2526926, GB2418812, MY134217
- US7499441, AU2005262626, CA2570441, GB2430333, JP4690397, NZ551231, KR100869043, RU2351080, BRP10512052, MXPA06015241
- US 7200129, AU2005226531, CA2558551, CN1930809, GB2429378, JP4351720, BRP10508671, NZ549124, KR848483, RU2321952

The AMBE+2™ voice coding Technology embodied in this product is protected by intellectual property rights including patent rights, copyrights and trade secrets of Digital Voice Systems, Inc. This voice coding Technology is licensed solely for use within this Communications Equipment. The user of this Technology is explicitly prohibited from attempting to decompile, reverse engineer, or disassemble the Object Code, or in any other way convert the Object Code into a human-readable form.

Environmental Responsibilities

Tait International Limited is an environmentally responsible company which supports waste minimization, material recovery and restrictions in the use of hazardous materials. The European Union's

Waste Electrical and Electronic Equipment (WEEE) Directive requires that this product be disposed of separately from the general waste stream when its service life is over. For more information about how to dispose of your unwanted Tait product, visit the Tait WEEE website at www.taitradio.com/weee. Please be environmentally responsible and dispose through the original supplier, or contact Tait International Limited.Tait International Limited also complies with the Restriction of the Use of Certain Hazardous Substances in Electrical and Electronic Equipment (RoHS) Directive in the European Union.In China, we comply with the Measures for Administration of the Pollution Control of Electronic Information Products. We will comply with environmental requirements in other markets as they are introduced.

Contents

Contact Information 2

Contents 3

Preface 4

1 Getting Started 6

2 Set Channel on Start-Up 10

3 Digital Inputs and Outputs 13

4 Select a Channel Using Digital Inputs 18

5 Drive a Digital Output Given an Alarm Condition 22

6 Transmit Lockout 25

7 High Availability Repeater 28

8 TaskBuilder Language 36

9 TaskBuilder Grammar 42

10 TaskBuilder Inputs and Actions 43

11 TaskBuilder Alarm names 45

12 Specifications and Limits 48

13 Change History 49

Preface

Scope of Manual

TaskBuilder allows the system designer to create rules that respond to base station conditions and events, and control base station operation. Examples of use include: select a channel based upon digital input states, drive a metallic hardware output on a defined alarm condition, and lock out the transmitter after a defined transmission time.

This manual introduces TaskBuilder through examples with explanations and provides a complete language reference.

It is intended to assist those who are responsible for designing, commissioning and maintaining systems. See Associated Documentation below for more specific information on base station configuration.

New in this Release

Release 3.25

`dig-out-13` is an output-only digital pin with the ability to sink current for driving a relay.

`trace:` statement allows programs to write directly to the log.

Web-UI: [Revert] button allows you to revert to the last good TaskBuilder program

See Change History for a complete description of all changes to TaskBuilder.

Alerts

This alert is used to highlight significant information that may be required to ensure that you perform procedures correctly, or to draw your attention to ways of doing things that can improve your efficiency or effectiveness.

Associated Documentation

The following associated documentation for this product is available on the Tait support website.

- MBC-00001-25 TB9400 Installation and Operation Manual
- MBC-00002-22 TB9400 Specifications Manual
- MBC-00008-25 TB9300 Installation and Operation Manual
- MBC-00009-22 TB9300 Specifications Manual
- MBD-00001-15 TB7300 Installation and Operation Manual
- MBD-00002-16 TB7300 Specifications Manual
- MNB-00010-06 DMR Channel Group System Manual
- MND-00001-10 AS-IP Channel Group System Manual
- MND-00002-08 P25 Channel Group System Manual

Publication Record

Issue	Publication Date	Description
3	October 2021	Updated with v3.25 content.
2	April 2021	Updated with v3.20 content.
1	March 2021	First release. V3.15.

1 Getting Started

This chapter introduces TaskBuilder and provides a basic explanation of how to work with TaskBuilder programs.

1. Introduce TaskBuilder
2. What does 'running a TaskBuilder program' mean?
3. Get a script onto the base station.
4. How to see if it is running?
5. Simple program: Use a timer to toggle a digital output.
6. It didn't do what I expected!

1.1 Introducing TaskBuilder

TaskBuilder provides a way to control base station operation.

1.1.1 TaskBuilder allows the base station to:

- Respond to alarms
- Raise and clear custom alarms
- Respond to digital inputs
- Sense base station Tx
- Activate digital outputs
- Change the base station channel

TaskBuilder programs specify actions to take when given conditions are true and when specified events occur. Here is an example that sets the channel when the base station goes online:

```
// TaskBuilder Example 1
when:  operation.running then: channel => 2
```

Double slash (//) denotes a comment. Comments can appear on a line following TaskBuilder statements, or on a line by themselves. The TaskBuilder compiler ignores comments, so you can use them freely to remind yourself what the program is supposed to do.

1.2 TaskBuilder feature license

To use TaskBuilder you must have a TBAS073 TaskBuilder License. Contact Tait for more information.

1.3 Running a TaskBuilder program

The base station Web UI has two pages that allow you to control the operation of TaskBuilder and monitor its execution. See Monitoring TaskBuilder executionfor information about monitoring.

The Tools > TaskBuilder page allows you to manage TaskBuilder programs, including:

- Display TaskBuilder status
- Start and stop TaskBuilder execution
- Choose whether TaskBuilder should run when the base station goes online
- Display the current TaskBuilder program text
- Edit the TaskBuilder program
- Display program errors, if any are present
- Read a TaskBuilder program from your computer
- Write a TaskBuilder program to your computer
- Revert to the last working TaskBuilder program if the current TaskBuilder program text contains errors

1.3.1 To run your first TaskBuilder program

In this exercise, you will run the following basic example on the Base Station and observe the result.

```
// TaskBuilder Example 1
when:  operation.running then: channel => 2
```

1. Set up channel 2 on the base station, or modify the program text above to reference a channel that is defined on the base station.
2. Ensure that your base station has a TaskBuilder feature license TBAS073.
3. Set the base station offline.
4. Go to the Tools > TaskBuilder page on the Web UI.
5. Type the program as shown above into the Program text area, and press **Save**.
6. Does the base station report it as good ? If not, see "It didn't do what I expected" below.
7. Set the base station online. If "Start TaskBuilder when going online" is checked, the Web UI should show TaskBuilder is running . If not, see "It didn't do what I expected" below.
8. Check one of the RF monitoring screens. It should show that the base station channel is 2, and that the reason is TaskBuilder.
9. Look in the TaskBuilder trace log (Monitor > TaskBuilder > Trace). You should see the execution of the rule in the log along with a timestamp.

1.4 Monitoring TaskBuilder execution

The TaskBuilder monitoring page on the base station Web UI (Monitor > TaskBuilder) allows you to verify that the TaskBuilder program is operating as expected.

- The Program states pane shows the states of your TaskBuilder program variables.
- The Trace pane shows log output in real time.
- You can choose what goes into the log. You can enable or disable log entries for triggered rules and trace actions (see below).

The Trace pane displays an execution history with two types of record:

1. Triggered rules, including the trigger (`when:` condition), and actions (`then:` condition). Triggered rules are useful as a record for what your TaskBuilder program did, and to confirm whether it is doing what you expect. The log of triggered rules can serve as a good test record.
2. User trace statements (when executed as part of an action). User trace statements give more targeted visibility into specific scenarios and conditions. If your program is not doing what you expect, you can apply trace statements as a 'trail of breadcrumbs' to locate the point where your program operation diverged

Executing the example program above produces a log message similar to the following :

```
2020-11-27T03:16:56.077170 rule: when: operation.running then: channel => 2
```

1.5 Trace actions

One TaskBuilder action allows you to trace the execution of your program with a message that you specify. The trace message is written to the output log, and can report the current values of TaskBuilder variables.

Here is the set channel program from the example above with a trace message added:

```
// TaskBuilder Example 2: include a trace message
when:  operation.running then:
    {
    channel => 2,
    trace: "channel is ${channel}"
    }
```

Running this program (with just user trace statements enabled) results in a log output similar to the following:

```
2021-08-29T23:49:34.170658 channel is 2
```

Tips:

It can be useful to enable logging of rules when first verifying the operation of a new TaskBuilder program. The logs will show when:
- the expected events occurred,
- the corresponding rules triggered,
- the rule actions were taken.

Once you are confident in program operation, disable the rules in the logs to keep the logs quiet and to minimize CPU overhead.

Add trace statements to your program when:
- Some aspect of your program is not doing what you expect and you want more visibility, or
- To provide a longer term record of important events and outcomes which is less verbose than dumping all triggered rules in the log.

1.6 It didn't do what I expected

If	Then
Program error	Check that the base station has the TaskBuilder license. Check your variables are spelled correctly. TaskBuilder keywords usually end in a colon: Did you forget to define your variable?
No visible result	Check the base station is online Try restarting TaskBuilder (Tools > TaskBuilder) Check that it compiled successfully (Tools > TaskBuilder) Does your script need a start-up trigger (`operation.running`)? Try adding a timer and digital I/O toggle and monitor the output.
Base station isn't doing what I expected	Did you read the correct program file onto the base station? Check the log file against the rules in your program.
Base station is unresponsive	Reset the base station, and take it offline on the Web UI. Unselect 'Start TaskBuilder when going online' (Tools > TaskBuilder). Review the trace information in Monitor > TaskBuilder.

2 Set Channel on Start-Up

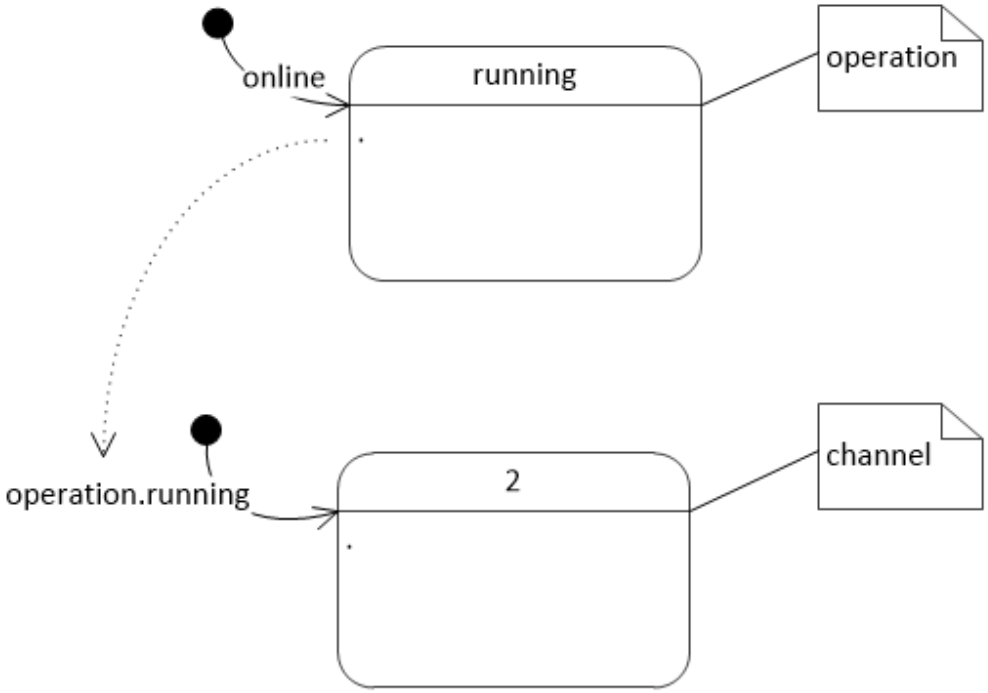
This chapter revisits the set channel examples and goes a bit more deeply into how to read it and recognize the different elements.

2.1 Example

Here it is again:

```
// TaskBuilder Example 1
when: operation.running then: channel => 2
```

The diagram provides a visualization of what happens:



The one-line TaskBuilder program relies on two variables:

`operation` is a standard TaskBuilder variable. It has a single state called `running`, which becomes active at the time when TaskBuilder starts - when the base station is taken online via the base station Web UI.

The other TaskBuilder variable is called `channel` which is described below.

The text '`when: operation.running then: channel => 2`' is called a `when:then:` rule. `when:then:` rules have the form:

`when:` event `then:` do action(s)

In this rule, the triggering event is `operation.running` which as described above, occurs when the base station goes online and TaskBuilder starts running. That is the primary purpose of the `operation.running` variable is to provide an initialization event when TaskBuilder starts.

When a rule is triggered by an event, TaskBuilder carries out the action(s) in the `then:` clause. The rule in this example has one action: `channel => 2.` This action requests the base station to change channel:

channel is a Standard Variable which represents the base station channel. TaskBuilder rules can react to the base station channel, and can change the base station channel.

`=>` is the 'become' operator. The become operator changes the state of a TaskBuilder variable.

This TaskBuilder program also includes a comment. Comments are good places to remind the reader what the program is for and provide contextual information.

```
// This is a comment
```

Double forward slashes can follow TaskBuilder statements on the same line:

```
when: operation.running then: channel => 2  // base station goes online.
```

2.1.1 State entry events

The example uses two standard TaskBuilder variables: `operation` and `channel`. It relies on some specific properties that all variables share:

1. In the example, the rule is triggered by the '`when: operation.running`' part. `operation.running` is an event corresponding to `operation` becoming `running`. This is a property of all TaskBuilder variables: When a variable enters a state, TaskBuilder generates an event with the name of that state.
2. There are three actions that can cause a variable to enter a state:

   • Base station operation. We saw that `2`is one possible `channel`. Whenever the base station changes channel the `channel` standard variable changes as well (and generates a state entry event).

   • Start up. All TaskBuilder variables have an initial state when TaskBuilder begins running. The initial state of TaskBuilder output variables, those writable by TaskBuilder (see TaskBuilder Inputs and Actions) is the first listed state in that document.

   • A become action (eg: `channel => 2`). Even if the target state is active already, the become operator causes that variable to (re) enter that state.

2.1.2 Revert channel on exit

TaskBuilder can respond to exit events as well as start up events. Suppose you want the base station to be on channel 2 when TaskBuilder is running, but should be on channel 3 otherwise. You can do this using the `operation.stopping` event:

```
// Base station is on channel 2 when TaskBuilder is running and on channel 3
otherwise


when: operation.running  then: channel => 2
when: operation.stopping then: channel => 3
```

On TaskBuilder exit (base station goes offline or user stops TaskBuilder from the Web UI), TaskBuilder will execute the actions for rules that include a `when: operation.stopping` clause. Those actions are the last to execute.

So in the example above, the base station will be on channel 3 after the base station stops. If the actions for that rule trigger further rules (such as `when: channel.3`), those further rules are not triggered.

2.1.3 Summary

What we learned

- TaskBuilder variables have well defined states.
- TaskBuilder generates state entry events when variables (re)enter a state. The name of the event is the name of the state (eg `operation.running`).
- `operation.running` is a start up event, which occurs when the variable `operation` enters its initial `running` state. It is a useful trigger for initialization actions.
- The `when:then:` rule specifies actions which should occur when triggered by the event specified in the `when:` condition.
- `channel` is a standard variable which can be changed using the become `=>` operator. The base station changes its channel in response to a `channel =>` become action.
- TaskBuilder recognizes everything following a double slash (`// This is a comment`) as commentary, and not part of the TaskBuilder program.
- `operation.stopping` is the last event to be processed when TaskBuilder stops. You can use it to set the base station to a well defined operating state.

3 Digital Inputs and Outputs

The first exercise in this chapter toggles a digital output at one second intervals. Then we extend the example so that the output only toggles if another input is low.

The exercises introduce the use of digital inputs and outputs, timers, user-defined variables and `given:when:then:` rules.

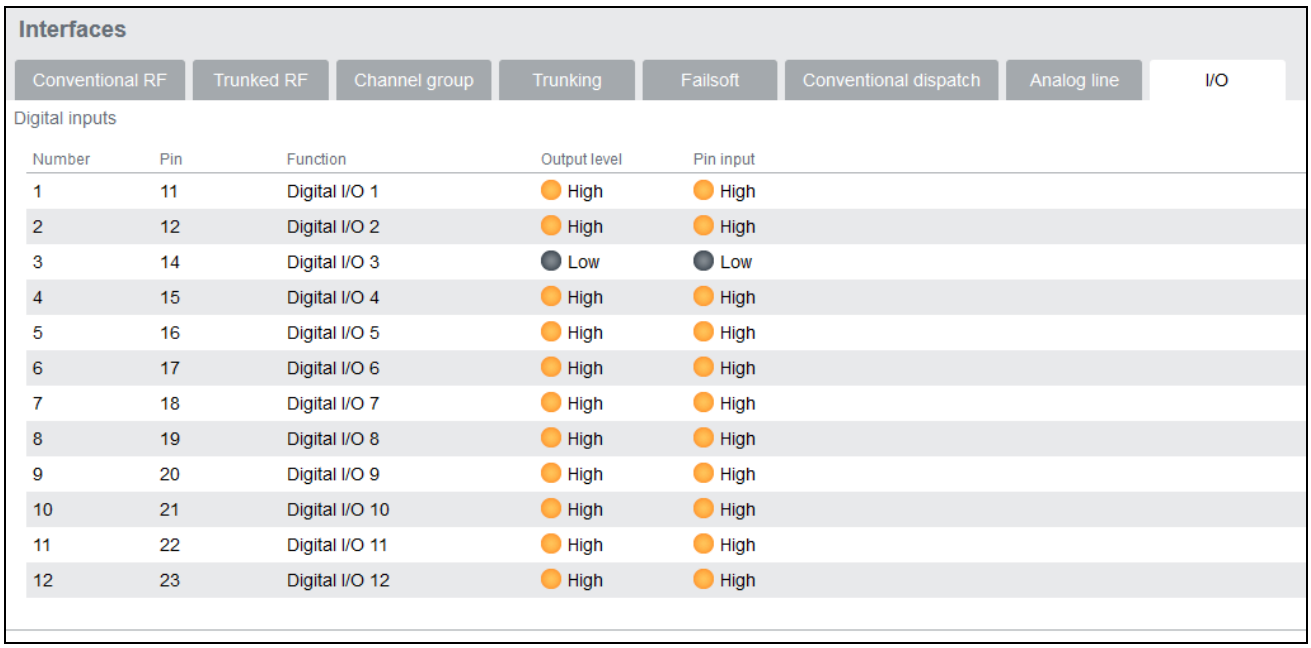
Toggling an output gives a visual indication that your TaskBuilder program is running, and could be a component of a high-availability (HA) set-up where one base station supervises another. The toggling output serves as a heart-beat.

The exercises in this chapter are set up so that you can enter the programs for yourself and see the results on the base station Web UI.

3.1 Digital I/O states

The base station has 12 digital I/O pins on the reciter DB-25 connector. (See the manuals for pinouts).

The Digital I/O monitoring page on the base station Web UI shows input and output states:

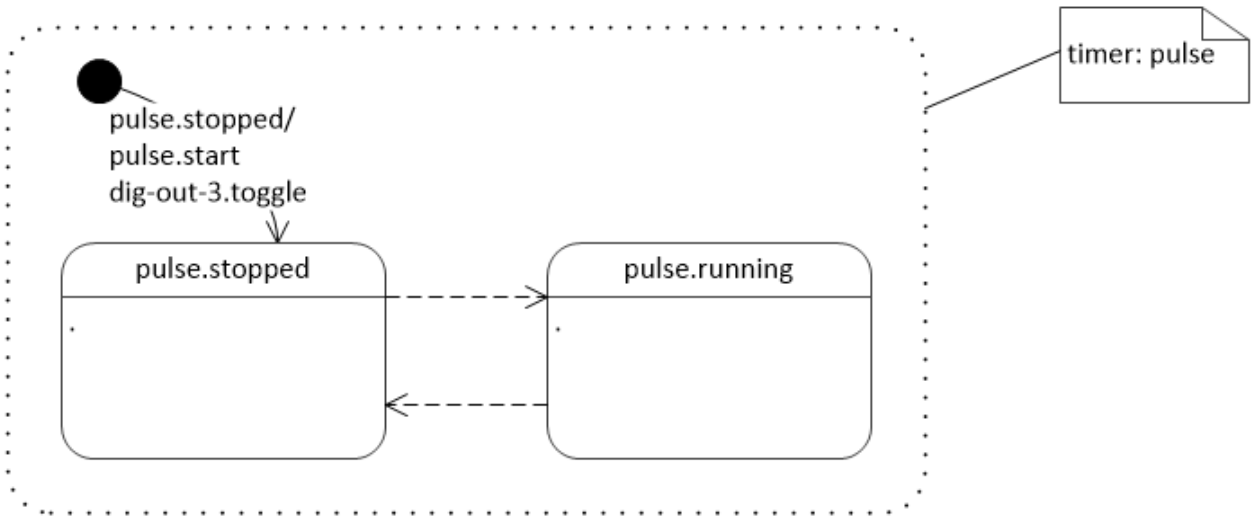


Number	Pin	Function	Output level	Pin input
1	11	Digital I/O 1	High	High
2	12	Digital I/O 2	High	High
3	14	Digital I/O 3	Low	Low
4	15	Digital I/O 4	High	High
5	16	Digital I/O 5	High	High
6	17	Digital I/O 6	High	High
7	18	Digital I/O 7	High	High
8	19	Digital I/O 8	High	High
9	20	Digital I/O 9	High	High
10	21	Digital I/O 10	High	High
11	22	Digital I/O 11	High	High
12	23	Digital I/O 12	High	High

We distinguish inputs from outputs because each pin can function simultaneously as an input and output. Each pin has an output transistor which can pull the voltage level low, and a pullup resistor that allows the input to float high. If the base station output or an external input pulls the pin low, then the voltage on the pin will be low (the screen-shot above shows the base station driving digital I/O 3 output low, with the input reading low as a result. No special initialization is required to use a digital I/O as an input. The digital output initialization values are high, which allows an external input to drive the pin into the desired logic state.

3.2 Using a timer to toggle the output

First the state transition diagram and program text, then the explanation breaks down how it works:




```
// toggle digital output 3 at one second intervals
timer: pulse interval: 1 :s
when: pulse.stopped then: { pulse.start, dig-out-3.toggle }
```

The program defines a `timer` called `pulse` and a single rule which, when the timer is stopped, toggles the digital output and starts the timer.

TaskBuilder timers are standard variables that your program can create and initialize with a timer interval. You can have intervals as integer numbers of milliseconds, seconds, minutes and hours (denoted `:ms, :s, :min, :hour` respectively).

This example uses a 1 second timer. Timers have two states, `stopped` and `running`. The initial state of a timer is `stopped`. Timers respond to a `start` event which starts the timer running. After the timer interval, the timer generates an `expired` event, and becomes stopped. The example responds to `pulse.stopped` which the timer generates on entering the `stopped` state.

The `when:then:` rule has two actions, `{ dig-out-3.toggle, pulse.start }`. To perform multiple actions, separate them with commas and surround the actions with curly brackets. Note that white space is not significant, so the rule could also have been written like

```
when: pulse.stopped then:
    {
    pulse.start,
    dig-out-3.toggle
    }
```

Breaking statements on multiple lines like this can be useful if you want to include a comment with some of the actions.

`dig-out-3` is a TaskBuilder standard variable. As a variable it can be assigned high and low states (become `=>` operator), and it can respond to a `toggle` event.

3.2.1 Raising events

`dig-out-3.toggle` and `pulse.start` are both events. You raise an event in a TaskBuilder action simply by writing the name of the event. Raising the event allows that event to trigger other rules in your program or base

station actions. We have already seen that timers respond to a `start` event by going to their `running` state. Digital outputs respond to a `toggle` event by changing the output state from high to low or vice-versa.

TaskBuilder offers only two types of action. The become `=>` operator changes the state of a TaskBuilder variable. Raising an event can trigger other rules in your TaskBuilder program, or cause a pre-defined action in a standard variable (as is the case in this example).

3.2.2 Summary

What we learned in this exercise:

- Digital I/Os are bidirectional - TaskBuilder can drive outputs and respond to inputs.
- Programs create instances of timer standard variables. Timers have intervals in the range of milliseconds to hours.
- The `then:` clause of a `when:then:` rule can have a list of actions. The actions are separated by commas, and the list of actions is surrounded by curly brackets `{ }`.
- You can freely use white-space in formatting TaskBuilder programs.

3.3 Toggle an output only when digital input 4 is low

What if a digital output should only toggle when digital input 4 is low? This exercise introduces:

- stateful behaviors
- digital output 13 is an output-only pin, with current sinking capabilities suitable for driving a relay

Digital output 13 is an output-only pin. TaskBuilder programs can change and respond to `dig-out-13`, but cannot reference the corresponding `dig-in-13`

The state transition diagram shows what we want:

The program logic has two states. When digital input 4 is high, the no-toggle state does nothing interesting. When digital input 4 is low, a toggling state uses a timer to toggle digital output thirteen.

The equivalent TaskBuilder program matches the diagram:

```
// Toggle digital output 13 at one second intervals
// but only when digital input 4 is low

timer: pulse interval: 1 :s
    // pulse control defines whether to toggle the output pin.

active: pulse-control has-states: { no-toggle, toggling }
when: dig-in-4.high           then: pulse-control => no-toggle
when: dig-in-4.low            then: pulse-control => toggling

    // toggling
when: pulse-control.toggling  then: { pulse.start, dig-out-13.toggle }

given: pulse-control.toggling
    when: pulse.expire        then: pulse-control => toggling
```

The free use of white-space visually aligns the different parts of each rule and quickly pick out what is relevant when reading the program.

Breaking the program down:

```
// pulse control defines whether to toggle the output pin.
active: pulse-control has-states: { no-toggle, toggling }
```

This line creates an active variable called `pulse-control` which can have two states: `no-toggle` and `toggling`. Making the states explicit lets us divide the problem in two parts:

1. What is the logic for deciding the state of `pulse-control`, and
2. What are the behaviors wanted for each distinct state.

The rules for defining which state is active are straight-forward to express:

```
when: dig-in-4.high           then: pulse-control => no-toggle
when: dig-in-4.low            then: pulse-control => toggling
```

`pulse-control` becomes `no-toggle` or `toggling` when a change in `dig-in-4` is detected.

The `no-toggle` state has no interesting behaviors, so it has no rules.

The `toggling` state should start the timer and toggle the digital output, as in the previous example:

```
when: pulse-control.toggling then: { pulse.start, dig-out-13.toggle }
```

Lastly, we need a rule for the expiry of the pulse timer. We already have a rule for `pulse-control.toggling` to perform those actions - so all that is necessary is to re-initialize `pulse-control`:

```
given: pulse-control.toggling when: pulse.expire then: pulse-control => toggling
```

This TaskBuilder statement adds a `given:` condition to the `when:then:` rule. The rule is only triggered if `pulse-control` is in the `toggling` state (because we only want to toggle the output in that state).

Because the `pulse-control.toggling` state already has a definition for the actions to perform on entering that state:

```
when: pulse-control.toggling then: { pulse.start, dig-out-13.toggle }
```

it is sufficient to simply re-enter the state when the timer expires (`pulse-control => toggling` above).

3.3.1 Active states and the given:when:then rule

The `given:` clause of a `given:when:then:` rule must always be the fully qualified name of a state. A fully qualified state name is written in the form `variable-name.state-name` (such as `pulse-control.toggling`).

State names can be user defined active variables (as in this example), standard variable states (such as `dig-in-4.low`) or composite variable states (which are introduced in later examples).

The `given:when:then:` rule performs the specified actions `when:` the specified event occurs but only if the `given:` nominated state is active.

4 Select a Channel Using Digital Inputs

The exercises in this chapter show two approaches for selecting a channel based upon states of the base station digital inputs. It also introduces composite states which allow you to express combinations of states.

4.1 Using 2 inputs to select 2 channels

Exercise: Write a program to select channel 20 on digital input 3 low and channel 21 on digital input 4 low.

The program is quite simple using what we have already learned:

```
// Select channel 20 on dig-input 3 low, and channel 21 on dig-input 4 low
when: dig-in-3.low then: channel => 20
when: dig-in-4.low then: channel => 21
```

You can generalize this simple example to more channels by adding more inputs and more rules.

Some questions to think about are:

- What channel will the base station be on if both inputs are low?
- Is it possible for digital inputs 3 and 4 to be high and low respectively, yet the base station is on channel 20?

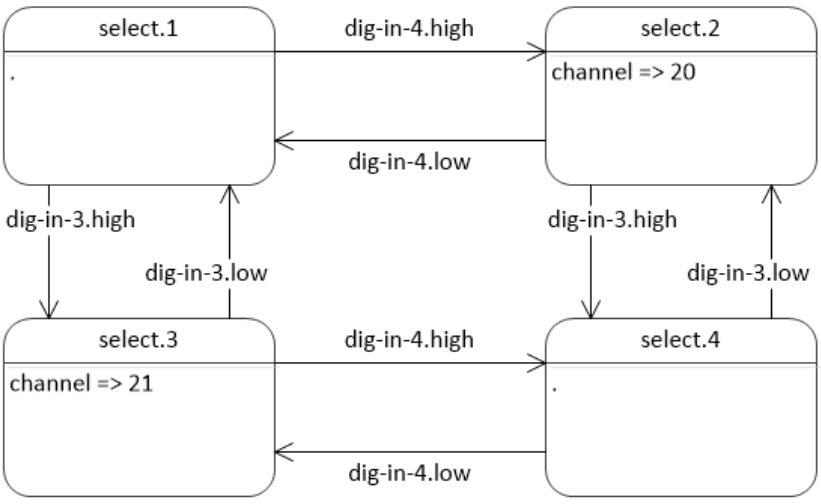
If you are using a rotary switch to select the base station channel, this approach is probably fine. Rotary switches typically have a 'break before make' characteristic, so should not suffer from multiple inputs being low at the same time.

If you are using a switch or switches that could create intermediate states, you probably want logic something like:

Select channel 20 when digital input 4 is high AND digital input 3 is low.

Select channel 21 when digital input 4 is low AND digital input 3 is high.

How to create combinations of states? You could create a state variable and cover all of the possible states and transitions of digital inputs 3 & 4:



This is complex, easy to get wrong, and does not scale.

4.2 Composite states

Composite states allow you to define states as combinations of other states. Here is a program that selects channel 20 or 21 depending which of digital inputs 3 and 4 are low:

```
// Select channel 20 on digital input 3 low, and 21 on digital input 4 low.
composite-state: select.0 = dig-in-4.high AND dig-in-3.low
composite-state: select.1 = dig-in-4.low  AND dig-in-3.high

when: select.0 then: channel => 20
when: select.1 then: channel => 21
```

With this program it is still possible for the base station to be on either channel 20 or 21 when both inputs are high or low (as in the previous example), but unlike the last example, if digital inputs 3 & 4 are high and low respectively, the base station will always go to channel 21.

4.3 Using 2 inputs to select 4 channels

Composite states make it easy to use combinations of input conditions to select channels. With two inputs it is possible to select between four channels:

```
// Select channels 20 to 23 based on combinations of digital inputs 3 & 4.
composite-state: select.0 = dig-in-4.low  AND dig-in-3.low
composite-state: select.1 = dig-in-4.low  AND dig-in-3.high
composite-state: select.2 = dig-in-4.high AND dig-in-3.low
composite-state: select.3 = dig-in-4.high AND dig-in-3.high

when: select.0 then: channel => 20
when: select.1 then: channel => 21
when: select.2 then: channel => 22
when: select.3 then: channel => 23
```

4.4 Debouncing the switch input

The final example uses a timer to add debounce to the inputs. Switch contacts can bounce, creating intermediate transient states which TaskBuilder can react to, resulting in multiple channel change requests in a short time. The base station internals are robust, but it is good practice to select a channel only when the switch contacts have settled. The program starts a debounce timer when any input changes, and only selects the target channel once the timer expires.

```
// Select channels 20 to 23 based on combinations of digital inputs 3 & 4.
// Change channel only when the debounce timer expires.


timer: debounce interval: 50 :ms
when: dig-in-3.change then: debounce.start
when: dig-in-4.change then: debounce.start


composite-state: select.0 = dig-in-4.low  AND dig-in-3.low
composite-state: select.1 = dig-in-4.low  AND dig-in-3.high
composite-state: select.2 = dig-in-4.high AND dig-in-3.low
composite-state: select.3 = dig-in-4.high AND dig-in-3.high


given: select.0 when: debounce.expire then: channel => 20
given: select.1 when: debounce.expire then: channel => 21
given: select.2 when: debounce.expire then: channel => 22
given: select.3 when: debounce.expire then: channel => 23
```

`change` is an event generated by digital inputs when the input state changes. If the inputs have contact bounce, the debounce timer may be (re)started multiple times, but it will only expire 50 ms after the last restart of the timer. The program uses `given:when:then:` rules to select the correct channel when the debounce timer expires.

4.5 More on composite states

Composite states can be defined in terms of any other states, including the states of standard variables, active variables or other composite states.

Composite state definitions can use boolean operators OR, AND, NOT and parentheses (). You can freely mix them in an expression as you would for an arithmetic expression. Like arithmetic expressions, the boolean operators have a precedence order: `()` > `NOT` > `AND` > `OR`.

So, for example if `var-1.true` is a state that is currently active and `var-2.false` is not currently active, then `NOT (var-1.true OR var-2.false)` gives a different result to `(NOT var-1.true OR var-2.false)`

Comparison between active variables and composite states

Both active variables and composite states give you a way to define the conditions for responding to an event (when used in a `given:` clause) and as a trigger (the `when:` clause). There are some differences as well:

- Active variables define a set of states, of which exactly one is true at any time.
- Composite states do not have any specific relationship to each other. Given a set of composite states with the same variable name (eg select in the example above), none may be active at any given time, or multiple may be active. (For example, write down a truth table to convince yourself that `var.a OR var.b` is active whenever `NOT(NOT var.a AND NOT var.b)` is.
- Variable names: Although composite states can share a variable name, it is for notational convenience only. Although the `select` states in the example have an obvious real-world relationship, TaskBuilder treats them as independently defined states.
- The become => operator applies to active variables but not composite states. The value of a composite state depends only on the states from which it is derived. Applying the become operator to a composite state is a program error.
- Composite states are good for capturing combinations of input conditions. Active variable states are useful to represent distinct sets of behaviors.

4.6 State diagrams

The examples in this chapter mostly don't have state diagrams. Given that TaskBuilder treats composite states independently, while it is possible to draw state diagrams including composite states (see the example above), there is not the same direct relationship between diagram elements and program elements as there is with state diagrams based on active variables. Again, this goes to the different uses for active variables versus composite states: If the problem domain and solution are naturally expressed using a state transition diagram, then there is likely a straightforward solution using active variables. If the problem space suffers from a potential explosion of states, then composite states may reduce that problem space complexity.

5 Drive a Digital Output Given an Alarm Condition

A common requirement is to drive a contact closure if an alarm is present. The contact could be connected to a warning light at a local or remote location.

The examples in this chapter include:

- Light a lamp when the base station front panel is absent.
- Indicate a major alarm.

The base station has many alarms, all of which can be used as TaskBuilder inputs. The complete set of alarms is listed here. TaskBuilder treats alarms as individual state variables, each of which can have the values of `active`, `inactive` or `disabled`.

5.1 Light a lamp when the base station front panel is absent

The front panel includes the fans that cool the base station and allow it to operate under the widest range of temperatures. The front panel may be removed to replace modules, but forgetting to replace the front panel could require a costly return to site. This example uses TaskBuilder to drive digital output 10 low, to close a contact and light a lamp when a base station has the front panel removed.

The specifications manuals listed in Associated Documentation define the operating current and voltage conditions for the digital inputs and outputs. Using a digital output to drive a relay may require an electrical interface circuit.

```
// Drive digital output 10 low when the front panel alarm is active
when: alarm-front-panel-not-detected.active then: dig-out-10 => low
```

This is not quite the end of the story, because you probably also want the light to turn off when the alarm is no longer active. You can do that in different ways - use multiple rules:

```
// Drive digital output 10 low when the front panel alarm is active - variation 1
when: alarm-front-panel-not-detected.active   then: dig-out-10 => low
when: alarm-front-panel-not-detected.disabled then: dig-out-10 => high
when: alarm-front-panel-not-detected.inactive then: dig-out-10 => high
```

Or, you could use a composite state:

```
// Drive digital output 10 low when the front panel alarm is active - variation 2
composite-state: fp-alarm.not-active = NOT alarm-front-panel-not-detected.active


when: alarm-front-panel-not-detected.active then: dig-out-10 => low
when: fp-alarm.not-active                  then: dig-out-10 => high
```

The first variation is simpler, and more flexible, but you may forget to account for the disabled state; using composite states is also the natural way to combine alarm inputs. See the next example.

5.2 P25 major alarm

DMR base station firmware assigns to alarms a status: Under a major alarm condition the channel is unusable. With a minor alarm the channel may be degraded but still provide service. The result of the major alarm could be a remote status indication, or take the channel out of service, or even switch another channel into service as a replacement.

This example provides the equivalent of the DMR major alarm with P25 firmware on a TB7300. The specific conditions that contribute to a major alarm are of course system specific. For simplicity, this example uses the same values as the DMR fixed and default configurable values.

The DMR major alarms (using default values where configurable) on TB7300 are:

PA calibration invalid, PA shutdown, 1PPS pulse absent, Channel invalid, Simulcast unsynchronized, Receiver calibration invalid, Hardware configuration invalid, 25 MHz synthesizer out of lock, 61.44 MHz synthesizer out of lock, TxF synthesizer out of lock, Rx synthesizer out of lock

The standard TaskBuilder alarms are listed here.

It is straightforward to define a major alarm as a TaskBuilder composite state:

```
// Major alarms result in the base station being out of service
composite-state: major-alarm.active =
    alarm-pa-calibration-invalid.active            OR
    alarm-pa-shutdown.active                       OR
    alarm-1pps-pulse-absent.active                 OR
    alarm-simulcast-unsynchronized.active          OR
    alarm-receiver-calibration-invalid.active      OR
    alarm-hardware-configuration-invalid.active    OR
    alarm-25-mhz-synthesizer-out-of-lock.active    OR
    alarm-61-44-mhz-synthesizer-out-of-lock.active OR
    alarm-txf-synthesizer-out-of-lock.active       OR
    alarm-rx-synthesizer-out-of-lock.active
```

To drive a pin, we want the inactive alarm state as well:

```
composite-state: major-alarm.inactive = NOT major-alarm.active
when: major-alarm.active   then: dig-out-10 => low
when: major-alarm.inactive then: dig-out-10 => high
```

5.3 Raising a custom alarm

Extending the example above, it is possible to assert a custom alarm for a given TaskBuilder condition (see TaskBuilder inputs and actions). To associate Custom alarm 1 with the major alarm from the example above , we would add rules such as:

```
when: major-alarm.active   then:
    { dig-out-10 => low,  alarm-custom-alarm-1.raise }
when: major-alarm.inactive then:
    { dig-out-10 => high, alarm-custom-alarm-1.clear }
```

5.4 Summary

What we learned:

- When writing rules that react to the presence of an alarm (such as driving a contact output), ensure that you capture the conditions both for asserting the pin output and de-asserting. You can either provide rules for all the alarm states (including disabled) or use a composite state.
- Composite states are a good way to represent combinations of alarms.
- TaskBuilder can raise and clear custom alarms.

6 Transmit Lockout

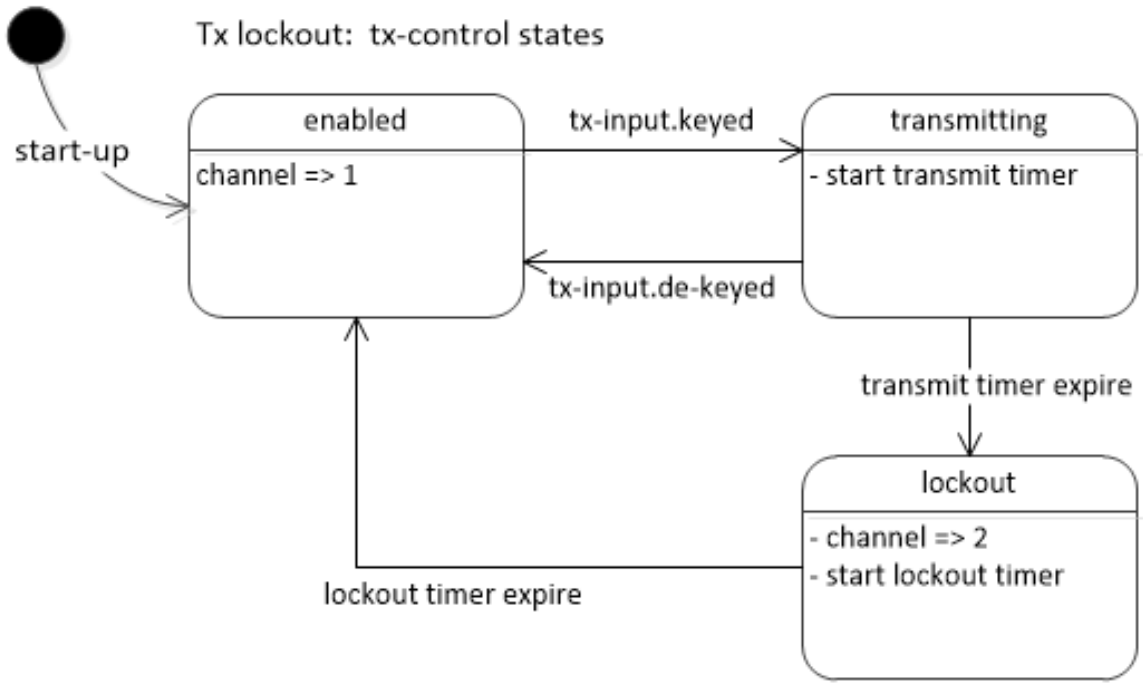
This chapter shows a transmit lockout use case with graphical and TaskBuilder solutions and offers some advice for good program style.

6.1 Tx lockout example

Problem

Solar and battery powered sites may provide a transmit lockout function to preserve battery storage. After transmitting for a maximum time, stop transmitting and wait until the transmitter is no longer keyed before resetting the lockout condition. Re-enable the transmitter once it is no longer keyed.

Here is a state transition diagram illustrating a tx-control function implementing lockout. Channel 1 is able to transmit. Channel 2 has transmit disabled.



In the enabled state the repeater will transmit if keyed.

In the transmitting state, the base station is keyed, and the transmit timer will end the transmission if it expires.

In the lockout state, the transmitter is disabled and a lockout timer is running.

Lockout ends only when both the lockout timer has stopped (expired) and the transmit input has finished.

Here is a TaskBuilder program that does the same thing. It is the same design: The TaskBuilder elements correspond to the diagram.

```
// Transmit lockout.
// After 30 seconds of transmission, lock out the transmitter:
// Useful for solar or battery operation.
// To re-enable the transmitter, the lockout timer must complete,
// and the transmit source must be removed.
// 2020-12-11 Tait Communications, Iain McInnes

// Tx control enables or disables transmission based on a timeout.
active: tx-control has-states:
    {
    enabled,      // would transmit if keyed.
    transmitting, // is transmitting - waiting for lockout.
    lockout,      // transmission was too long - wait for timer
                  // and end of transmission request
    }

timer: tx-timer      interval: 30 :s
timer: lockout-timer interval: 10 :s

// Condition for ending lockout
composite-state: lockout.end = lockout-timer.stopped AND tx-input.de-keyed

// enabled state
when:  tx-control.enabled      then: channel    => 1 // channel 1 allows tx
given: tx-control.enabled
when:  tx-input.keyed          then: tx-control => transmitting

// transmitting state
when:  tx-control.transmitting then: tx-timer.start

given: tx-control.transmitting
when:  tx-input.de-keyed       then: tx-control => enabled

given: tx-control.transmitting
when:  tx-timer.expire         then: tx-control => lockout


// lockout state
                                    // channel 2 has tx disabled
when:  tx-control.lockout      then: { channel  => 2, lockout-timer.start }
given: tx-control.lockout
when:  lockout.end             then: tx-control => enabled
```

6.1.1 Implementation notes

`tx-input` is a standard variable that reports whether the base station would be transmitting if it was able (transmit signal is present). The most likely reason not to be able to transmit is the channel configuration has transmit operation disabled (Configure > RF Interfaces > Channel profiles > Transmitter enabled).

The composite state `lockout.end` is a good example of simplifying the solution when it depends on multiple inputs:

```
composite-state: lockout.end = lockout-timer.stopped AND tx-input.de-keyed
```

The rule uses the timer stopped state rather than the `expire` event because composite states are derived from states, not events.

You could achieve the same thing with two extra `tx-control` states (perhaps labeled `lockout-timeout` and `tx-dekeyed`) and a few additional rules. The composite state expresses the solution intention better.

6.2 Good TaskBuilder style

This example (and the others in the user guide) are written in a particular style. The style mimics a state transition diagram (which the examples also provide where useful).

The benefits of using a specific style are solutions that are more predictable, without losing any expressiveness. They will be easier to design, to read, to pick-up defects, and which communicate better their intention. All of these things result in solutions that are more likely to be correct and do what you want.

Style recommendations are as follow:

1. A few lines of comment at the beginning explain the purpose and provide a brief summary of the operation of the program. For a working program (as opposed to a toy exercise) it is useful for the comment block to identify the author and date.

2. Use active variable states to capture individual required system behaviors.

3. Use composite variables to capture combinations of input conditions

4. Give state variables a well defined purpose. Active variable declarations should have a single-line comment stating the purpose:

```
// Tx control enables or disables transmission based on a timeout.
```

5. When defining states (active and composite), each state has a brief comment summarizing the system condition or behavior that the state represents.

6. Use good, descriptive state and variable names. You know you have succeeded when the program rules read naturally.

7. States will often have a timer that is associated with that state (waiting for timeout). The timer is started as a state entry action, and provides a default exit event for the state.

8. Rules are organized by variable states, with a single line comment grouping the rules associated with that state.

9. Where possible, have (changing channel, starting timers) actions occur on entry to the state that is associated with those actions:

```
when: tx-control.lockout then: { channel => 2, lockout-timer.start }
```

10. The subsequent rules for each state have the state name as a given: condition. The action associated with the rule should be a simple state transition:

```
given: tx-control.transmitting when: tx-timer.expire then: tx-control => lockout
```

11. Use vertical alignment to make it easy for the eye to distinguish distinct rule elements (`when:`, `then:` and actions)

12. Where useful, additional comments add context to rules. Use rule comments sparingly. If a rule needs a comment to explain it, then ask whether the states are well partitioned and states and events are well named. See recommendation 5.

7 High Availability Repeater

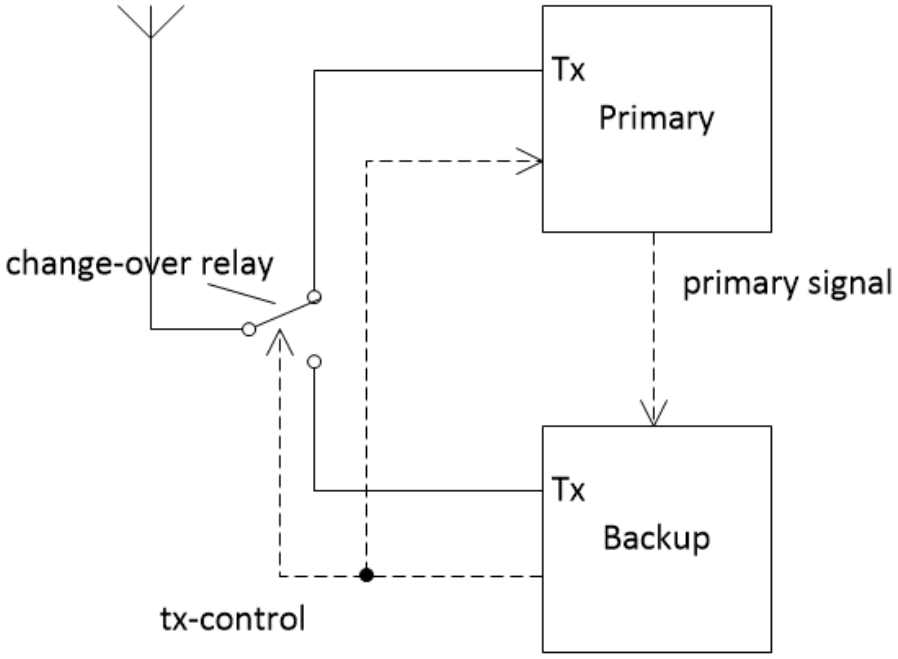
7.1 Requirements

If a channel has stringent down time requirements, the channel can be made resilient to the failure of a repeater by deploying primary and backup repeaters: If the primary repeater is not able to provide service, the backup repeater takes over.

- A major alarm or power fail results in a 'no service' condition. The repeater can not provide service.
- The solution uses a change-over relay to switch the transmit antenna to the primary or secondary repeater. The solution needs to produce a signal that drives the relay, and must avoid operating the relay when either base station is transmitting.

7.2 Problem logic

The diagram shows a possible solution:



The backup supervises the primary by means of a digital signal from the primary, which indicates whether the primary is up (nominal) or down (failed).

The backup has the decision logic for which repeater is in service. It outputs a tx-control signal which drives a change-over relay and informs the primary which repeater is in service.

Primary operation has the following states:

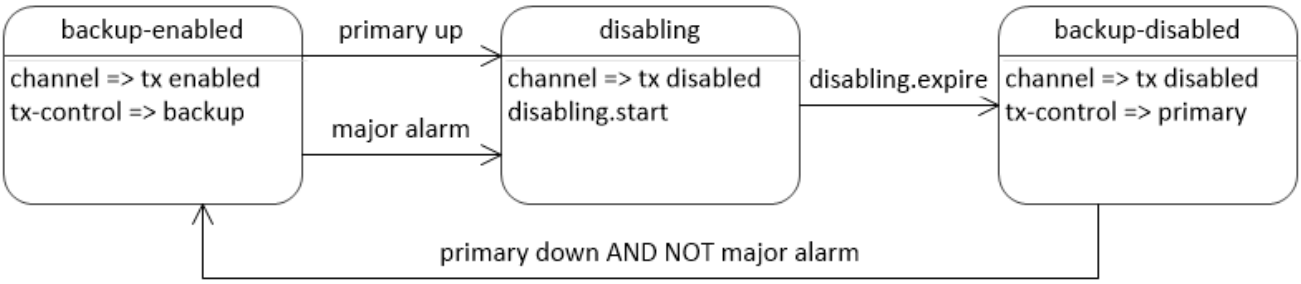
State	Composite inputs	Outputs	Commentary
nominal	NOT major alarm AND tx-control is primary	primary signal => up. transmit => enabled.	Normal operating condition. Primary does not have a major alarm, primary is reporting an 'up' signal to backup, and backup has responded by switching the change-over relay to the primary.
down	major alarm	primary signal => down. transmit => disabled	Primary has a fault taking it out of service. It sets the primary output signal to 'down', and disables its transmitter. Backup is expected to operate the change-over relay.
recovering	NOT major alarm AND tx-control is backup	primary signal => up. transmit => disable	Primary has recovered from a fault. Primary is up, but primary can detect that the backup repeater still has the change-over relay in the backup position. Backup is expected to recognize that the primary is up and operate the change-over relay. Primary transmit is disabled.
offline	Caused by fault or user action	primary signal => down	Faults which cause the channel to be invalid also result in the base station being offline. TaskBuilder programs can respond to the `operation.stopping` event.

The primary states are composite (a combination of alarm status and tx-control input).

The primary should output a 'down' signal when it has a major alarm or when it goes offline. When the primary is outputting a down signal it must not transmit.

When the major alarm clears, it must output an 'up' signal, and wait until it sees that it has control (from the backup) before enabling transmit.

Backup operation has the following states



State	Composite inputs	Outputs	Commentary
backup-enabled	primary-signal.down AND NOT (backup) major alarm	tx enabled tx-control switched to backup	The backup switches the change-over relay if the primary is down and the backup is up.
disabling	primary-signal.up OR major alarm	disable transmit start disable timer	Allow for channel change time before switching the relay.
primary-enabled	disable timer expired	tx-control switched to primary	Otherwise the backup switches the change-over relay to the primary.
offline	Caused by fault or user action	tx-control switched to primary	Faults which cause the channel to be invalid also result in the base station being offline. TaskBuilder programs can respond to the `operation.stopping` event

Signal polarities

It is possible to choose signal polarities so as to make the system as robust as possible. If either the primary or backup is powered down or disconnected, the other should still operate.

- The primary signal indicates whether the primary is up or down. If it is unconnected or the primary is powered down, the input to the backup will float high. The backup should treat primary status as down when the signal input is high.
- The tx-control signal output from the backup drives the change-over relay and informs the primary. If the backup is disconnected or powered down, the input will float high at the primary. The primary should treat tx-control as primary when the input is high, and the relay should be in the primary position when it is de-energised.

Manual override

With the signal polarities as above, manually taking the primary or backup offline will result in the other one remaining in control.

Relay must not operate when either repeater is transmitting

The design assures this by means of the hardware handshake signals:

In the presence of a primary fault:

- The primary asserts the primary signal down and does not transmit
- The backup recognizes the primary is down, switches the change over relay to the secondary and can transmit.

When the primary fault clears:

- The primary asserts the primary signal up, but does not immediately transmit (recovering state).
- The backup recognizes the primary is up, stops any transmission itself, and switches tx-control to the primary.
- The primary recognizes that it has tx-control, and enables transmit once again.

7.3 TaskBuilder: Primary repeater definitions, states and logic

Commentary

For the primary repeater, all states are logical combinations of primary and backup operational status. The primary rules in the TaskBuilder solution just depend on the input conditions and their combinations.

Operation.stopping

The program responds to alarms and base station offline in different ways:

```
when: primary-operation.down       then: { signal    => down, channel => 2 }
when: operation.stopping           then: { dig-out-2 => high, channel => 2 }
```

Why isn't `operation.stopping` just a condition that contributes to `primary-operation.down` ?

`operation.stopping` is the last event to trigger any rule.

A rule such as:

```
when: operation.stopping           then: { signal => down, channel => 2 }
```

will not work as expected. It would depend upon the additional rule

```
when: signal.down then: dig-out-2 => high
```

before the output signal is actually asserted. But as discussed, since `operation.stopping` is the last event, `signal.down` is not able to trigger the rule that drives the output high.

The solution is to just drive the output high directly with the `operation.stopping` event.

Program text

```
// Simple high-availability solution utilizes two repeaters (primary and backup)
// The backup supervises the primary via a primary signal connection
// indicating the health of the primary (up or down)
// The backup operates a change-over relay (tx-control output)
// Primary or backup repeaters are down if they have a major alarm
// (see text below for definition of major alarm)
// Secondary must not operate relay while either repeater is transmitting.
// 2020-12-16 Tait Communications Iain McInnes
//---

//================================================
// Primary repeater definitions, states and logic.
//================================================
// Major alarms result in the base station being out of service
// Can customize for individual deployments.

composite-state: major-alarm.active =
    alarm-pa-calibration-invalid.active            OR
    alarm-pa-shutdown.active                       OR
    alarm-pa-forward-power-low.active              OR
    alarm-1pps-pulse-absent.active                 OR
    alarm-channel-invalid.active                   OR
    alarm-simulcast-unsynchronized.active          OR
    alarm-receiver-calibration-invalid.active      OR
    alarm-hardware-configuration-invalid.active    OR
    alarm-25-mhz-synthesizer-out-of-lock.active    OR
    alarm-61-44-mhz-synthesizer-out-of-lock.active OR
    alarm-txf-synthesizer-out-of-lock.active       OR
    alarm-rx-synthesizer-out-of-lock.active

composite-state: major-alarm.inactive = NOT major-alarm.active

//-----------------------------------------------
// Give the output health signal a readable name

active: signal has-states:
    {
    up,  // operation is nominal
    down // major alarm condition
    }
when: signal.up   then: dig-out-2 => low
when: signal.down then: dig-out-2 => high // will float high (down) if disconnect
                                          // or powered down.
```

```
//--------------------------------------------------
// Give the tx relay control signal a readable name

active: tx-control has-states:
    {
    primary, // relay switched to primary; can transmit
    backup   // relay switched to backup; must not transmit
    }

    // default to primary if input disconnected.
when: dig-in-1.high then: tx-control => primary
when: dig-in-1.low  then: tx-control => backup

//------------------
// Primary operation

    // primary is in service.
composite-state: primary-operation.nominal    =
    NOT major-alarm.active AND tx-control.primary

    // primary is out of service.
composite-state: primary-operation.down       = major-alarm.active

    // primary would be in service, but change-over relay is switched to backup.
composite-state: primary-operation.recovering =
    NOT major-alarm.active AND tx-control.backup

    // Channel 1 has transmit enabled.
when: primary-operation.nominal    then: { signal => up     , channel => 1 }


    // Channel 2 has transmit disabled.
when: primary-operation.down       then: { signal => down   , channel => 2 }
when: primary-operation.recovering then: { signal => up     , channel => 2 }

    // Signal down when go offline
when: operation.stopping           then: { dig-out-2 => high, channel => 2 }

//------------------
// End of program
```

7.4 Backup repeater definitions, states and logic

Commentary

The secondary repeater TaskBuilder program needs a timeout state: Waiting for the channel change to end any transmission before signaling the primary. The active variable `backup` captures these states of the backup.

The example below uses digital output 13 to control the relay and signal the primary base station. From firmware release 3.25, TaskBuilder can use the coax relay driver pin 24.
To drive a relay such as TBCA03-10 or TBDA03-10 your TaskBuilder program should use digital output 13.

Program text

```
// Backup repeater definitions, states and logic
// Simple high-availability solution utilizes two repeaters (primary and backup)
// The backup supervises the primary via a primary signal connection
// indicating the health of the primary (up or down)
// The backup operates a change-over relay (tx-control output)
// Primary or backup repeaters are down if they have a major alarm
// (see text below for definition of major alarm)
// Secondary must not operate relay while either repeater is transmitting.
// 2021-04-15 Tait Communications Iain McInnes
//---
//=============================================
// Backup repeater definitions, states and logic.
//=============================================
// Major alarms result in the base station being out of service
// Can customize for individual deployments.

composite-state: major-alarm.active =
    alarm-pa-calibration-invalid.active           OR
    alarm-pa-shutdown.active                      OR
    alarm-pa-forward-power-low.active             OR
    alarm-1pps-pulse-absent.active                OR
    alarm-channel-invalid.active                  OR
    alarm-simulcast-unsynchronized.active         OR
    alarm-receiver-calibration-invalid.active     OR
    alarm-hardware-configuration-invalid.active   OR
    alarm-25-mhz-synthesizer-out-of-lock.active   OR
    alarm-61-44-mhz-synthesizer-out-of-lock.active OR
    alarm-txf-synthesizer-out-of-lock.active      OR
    alarm-rx-synthesizer-out-of-lock.active

composite-state: major-alarm.inactive = NOT major-alarm.active

//-------------------------------------------------------------
// Give the input health signal from the primary a readable name
active: primary has-states:
    {
    up  , // operation is nominal
    down  // major alarm condition
    }

when: dig-in-2.low  then: primary => up
when: dig-in-2.high then: primary => down
```

```
//-----------------------------------------------------------
// States of the backup
active: backup has-states:
    {
    enabled  , // backup has taken control
    disabling, // changing channel to disabled TX
    disabled   // control given to primary
    }

timer: disabling interval: 1000 :ms        // wait for channel change to disable TX

//-----------------------------------------------------------
// Rules for backup

when: backup.enabled     then:
    { dig-out-13 => low, channel => 1 }            // enable TX when backup enabled

when: primary.up         then: backup => disabling // hand control back to primary
when: major-alarm.active then: backup => disabling

when: backup.disabling   then:
    { channel => 2, disabling.start } // disable TX and start the timer

when: disabling.expire   then: backup     => disabled  // finished changing channel
when: backup.disabled    then: dig-out-13 => high

//-----------------------------------------------------------------
// tx-control drives the change-over relay

composite-state: tx-control.backup = primary.down AND NOT major-alarm.active

when: tx-control.backup then: backup => enabled

when: operation.stopping then:
    { dig-out-13 => high, channel => 2 }     // Relinquish control when go offline

// End of program
```

8 TaskBuilder Language

This chapter defines how TaskBuilder programs are expressed and what they mean.

8.1 Syntax highlighting

In this document, TaskBuilder statements are written in a `fixed-width-font`

8.2 Names

TaskBuilder programs may assign names to active variables, states, events, and timers.

Names may include underscore_and-minus characters: `this_is-aValidIdentifier`

Names are case-insensitive. In base station firmware release 3.20, names are converted internally to lower case, and displayed (Web UI, logs) in lower case. In future releases, names will still be case-insensitive, but the original case of the input text will be preserved.

8.3 Keywords

TaskBuilder reserves the following keywords:

```
active: has-states: composite-state: and or not timer: interval: :ms :s :min :hour
given: when: then: raise trace:
```

TaskBuilder keywords are case-insensitive. They are displayed in lower case independently of the case in the input text.

8.4 Comments

Comments in TaskBuilder help the reader understand the meaning of a program. They have no effect on execution.

Comments begin with a double slash, and comprise any printable text up to the end of the line:

```
// This is a comment
```

Comments can appear on the same line as TaskBuilder statements:

```
when: operation.running then: channel => 2  // base station online.
```

8.5 Active variables

Active variables represent system state. Distinguishing states allows TaskBuilder to do different things in response to an input event, depending on the state.

An active variable is a set of mutually-exclusive states:

```
active: a-variable has-states: { state-1, state-2, state-3 }
active: beacon     has-states: { on, off }
```

When an active variable has a value of a given state, we say that state is active. In a TaskBuilder program, an active state is written with a 'dot' notation - all of the following are true or false depending on whether the respective state is active:

```
a-variable.state-1
operation.running // base station online.
beacon.off
```

The initial value of an active variable when TaskBuilder starts is the first listed state.

8.6 Composite variables

A composite state is defined as a logical combination of other states:

```
composite-state: alarm.major =
    alarm-PA-forward-power-low.active           OR
    alarm-PA-reverse-power-high.active          OR
    alarm-PA-vswr-high.active                   OR
    alarm-25-MHz-synthesizer-out-of-lock.active


composite-state: alarm.minor =
    (
    alarm-PA-driver-temperature-high.active OR
    alarm-PMU-temperature-high.active       OR
    alarm-PMU-battery-voltage-low.active    OR
    alarm-PMU-mains-supply-failed.active
    )
    AND NOT alarm.major
```

As with active states, composite states notionally belong to a composite variable. The important differences between active and composite variables are:

1. The set of states belonging to a composite variable are not necessarily mutually exclusive, since they are defined by arbitrary logical functions. Therefore more than one state belonging to a given composite variable may be active at any time.
2. A TaskBuilder program may assign an active variable to become a given state, but may not assign a composite to become a given state, since the truthfulness of a composite state is given by the states from which it is derived.

Other than those differences, active and composite states behave the same way.

8.7 Timer variables

A timer variable is defined using the statement:

`timer: timer-variable-name interval: expiration-time`

Timers have states: `{ stopped, running }`

Timers begin `running` when they are sent a `start` event.

Sending a `start` event to a running timer restarts the timer.

After the expiration time following the last `start` event, the timer produces an `expired` event and becomes `stopped`

8.8 Standard variables

A timer is an example of a TaskBuilder standard variable. Standard variables have pre-defined names and behaviors. TaskBuilder Inputs and Actions defines the variables making up the TaskBuilder standard library.

8.9 TaskBuilder Rules

8.9.1 The when:then: rule

The `when:then:` rule is written

```
when: variable.event then: action
```

or

```
when: variable.event then: { action-1, action-2, ... }
```

The rule defines the actions to occur in response the given event, for example:

```
when: debounce.expire then: channel => 1 // go to channel 1 after debounce time.
```

The => symbol is the 'become' operator, see below.

8.9.2 The given:when:then: rule

The `given:when:then:` rule is written

```
given: var-1.state when: var-2.event then: action
```

or

```
given: var-1.state when: var-2 then: { action-1, action-2, ... }
```

The `given:when:then:` rule adds an additional condition to the `when:then:` rule. TaskBuilder only performs the listed actions if the given state is active when the event occurs.

The state referenced by the `given:` clause of the `given:when:then` rule is always a fully qualified state name; that is, it follows the form `variable-name.state-name`

Rule terminology

`when:then:` rules are always triggered when the event in the rule's `when:` clause occurs. `given:when:then:` rules are triggered when the rule state is also active when the rule event occurs. Triggering a rule causes TaskBuilder to carry out the rule actions.

8.10 Events

Events are triggers that cause actions to occur. Events are generated by a `raise` action. Events carry no information other than their identity (and implicitly, their time sequence).

Events don't have to be explicitly defined; `when:` and `then:` clauses make it clear when a name should refer to an event.

8.11 State entry events

Good TaskBuilder practice is to identify and define states that represent characteristic behaviors, since that is how people think about states. To assist that idiom, TaskBuilder allows a `when:` clause to refer to a state name. The rule will be triggered when the variable enters that state:

```
when: var.state then: { actions-associated-with-a-state }
```

A `state-name` appearing in a `when:` clause refers to the event that is raised automatically on entering that state.

Events can be used without having to be declared. An example fragment is:

```
active: a-variable  has-states: { state-1, state-2, state-3 }
                          when: some.condition  then: raise a-variable.next
given: a-variable.state-1 when: a-variable.next then: a-variable => state-2
```

Event lifetime

The lifetime of an event is the time between the event being raised, and the time when all rules referencing the event may be triggered (a rule may not be triggered if the `given:` clause is not satisfied at the time of the event). Events are atomic, and can be queued if events are occurring more quickly than the associated rules can be matched and executed. It is possible for multiple instances of the same event to be queued waiting to trigger their associated rules. In the most severe case, events may be discarded if it is not possible to enqueue the event.

8.12 Actions

TaskBuilder rules have either a single action:

```
when: variable.event then: action
```

or a list of actions:

```
when: variable.event then: { action-1, action-2, ... }
```

In the case of the list, the actions are carried out in order of the listed sequence.

Possible TaskBuilder actions are raise event(s) and change state(s):

8.12.1 Raise an event

Use the keyword `raise`, or just write the event:

```
raise dig-out-1.toggle
debounce.start
```

8.12.2 Become a (new) state

The become operator `'=>'` causes a TaskBuilder variable to change its state, if it is one that can be changed from TaskBuilder:

Variables that can be changed by a TaskBuilder program are: user-defined active variables, timers, standard library output variables.

Examples:

```
when: debounce.expired then: channel    => 1
when: alarm.minor      then: dig-out-5 => low
```

Composite variables and standard library inputs may not be written to using the become operator.

If a variable already has the same state as the target of the become operation, the existing state is re-entered, and TaskBuilder automatically raises associated state entry event.

Standard library input objects that generate inputs into TaskBuilder do not generally re-enter existing states. For example the rule,

```
when: dig-in-1.low then: do-stuff
```

will only be triggered on actual transitions of the input.

Similarly, asking standard library variables to become the same state they are already in will not necessarily cause any change to the base station. The following program does not necessarily cause the base station channel to be continuously re-initialized:

```
timer: repeating    interval: 1 :s
when: operation.running then: { channel => 1, repeating.start }
when: repeating.expire  then: { channel => 1, repeating.start }
```

8.12.3 Trace actions

One TaskBuilder action allows you to trace the execution of your program with a message that you specify. The trace message is written to the output log, and can report the current values of TaskBuilder variables.

Here is the set channel program from the example above with a trace message added:

```
// TaskBuilder Example 2: include a trace message
when:  operation.running then:
    {
    channel => 2,
    trace: "channel is ${channel}"
    }
```

8.13 Language design goals

The language here arose from the goals of:

1. Solve the same problems as Task manager

2. Orient the language around states and events: The 'active object' paradigm is robust, expressive, simple, well known, and fits base station internal execution.

3. Be expressive: The language should read naturally to people who are not expert TaskBuilder programmers without undue effort.

4. Be concise: Minimize the amount of stuff that is not directly involved in expressing the problem domain solution.

5. Minimize punctuation: because of (3) above.

6. Minimize declarations: because of (4) above. Events for example, are declared by using them.

7. Base stations mechanisms are expressed in the idioms of the language: Base station inputs and actions are defined in terms of TaskBuilder standard variables which interact with the running program in the ways described in this doc.

8.14 Comparison with Task manager

Task manager, on previous Tait base stations provides an equivalent facility to TaskBuilder.

- Active variables are the foundation of TaskBuilder, with states and events attached to active variables. Variables are active because the TaskBuilder design encourages you to associate behaviors (actions) with each distinct variable state.
- In TaskBuilder, the difference between states (which establish conditions for rules) and events (which trigger rules) is explicit.
- Composite states derive directly from Task Manager. User programs could synthesize the equivalent states explicitly using events, but it would be clumsy.
- TaskBuilder doesn't provide counters yet.
- Task Manager has a much greater range of inputs and actions.
- The design of TaskBuilder pays attention to the naturalness with which program rules can be articulated.
- Typing program text is still less intuitive and more error prone than selecting input rules and actions using the service kit UI.
- The performance envelope of TaskBuilder is controlled. The current rate of event processing is approximately 50 events per second (see the base station specifications manual).

9 TaskBuilder Grammar

This chapter defines the rules for a well formed TaskBuilder program.

Notation here is Wirth syntax notation.

Key to grammar

```
{ }                  - repeat zero or more times
[ ]                  - option
( stuff )            - group stuff
!                    - anything but
|                    - separate alternatives
"stuff"              - literal
<>                   - unprintable literal
```

Language clauses

```
TB-Logic-script       = statement { statement } .

statement             = active-declaration
                        | composite-declaration
                        | timer-declaration
                        | rule
                        | trace-statement .

active-declaration    = "active:" var-name "has-states:" qualifier-list .
qualifier-list        = "{" qualifier { "," qualifier } [,] "}" .
composite-declaration = "composite-state:" qualified-name "=" sum .
sum                   = product { "OR" product } .
product               = term { "AND" term } .
term                  = ["NOT"] ( qualified-name | "(" sum ")" ) .
timer-declaration     = "timer:" var-name "interval:" timer-interval .
rule                  = [ "given:" qualified-name ] "when:" qualified-name "then:"
                        (action | action-list) .
action-list           = "{" action { "," action } [,] "}"
action                = ["raise"] qualified-name
                        | [var-name] "=>" qualifier .
timer-interval        = digits (":ms" | ":s" | ":min" | ":hour") .
trace-statement       = "trace:" """ { printable-char | interpolated-variable } """
.
interpolated-variable = "${" var-name | qualified-name "}"
```

Language tokens

```
qualified-name        = var-name "." qualifier .
qualifier             = name | digits .
var-name              = name .
name                  = letter { name-char } .
name-char             = digit | letter | "-" | "_" .
digits                = digit {digit} .
comment               = "//" { !<newline> } .
```

10 TaskBuilder Inputs and Actions

What is it possible to do using TaskBuilder?

10.1 TaskBuilder standard variables

TaskBuilder inputs and actions are presented as a set of active variables each having distinct states and events. Whether a variable is primarily intended for input or output is a function of the events and states of that variable. The base station current channel for example is both an input and an output.

Notes to table:

1. All variables generate a state-change event when the respective state becomes active.
2. Variables with writable states (such as channel) can be changed from within TaskBuilder. They serve as TaskBuilder outputs.
3. All variables states are readable by TaskBuilder programs (can write rules that are triggered by the state condition). They serve as TaskBuilder inputs.
4. Alarm names are listed here: TaskBuilder: Alarm inputs and alarm names management.

10.2 List of TaskBuilder standard variables

Standard variable	Summary	Non-writable states	Writable states	Events accepted	Events generated
operation	Initialization and exit actions.	running - respond to start up stopping - respond to exit			
timer	Generate event after specified time		stopped, running	start - start the timer stop - stop the timer	expire - the time is up
dig-in-1.. dig-in-12	Digital inputs	high, low			change - the input state changed
dig-out-1.. dig-out-13	Digital outputs		high, low	toggle - change high to low and vice versa	
alarms (non-custom)[1]	State of the designated alarms	inactive, active, disabled			
alarms (custom)[1]	State of the designated alarms	disabled	inactive, active	raise, clear	
channel	Current base station channel		1 .. 1000	up, down	change
tx-status	Transmitting	de-keyed, keyed			
tx-input[2]	Request to transmit is present (analog or digital line interface, diagnostic test, CWID. tx-input is different from tx-status if transmission is prevented (disabled by configuration or TaskBuilder).	de-keyed, keyed			

[1]Alarm names are defined in TaskBuilder Alarm names

[2]The synchronized transmit is the only diagnostic test that can set tx-input to keyed since all the others take the base offline

11 TaskBuilder Alarm names

This chapter defines the alarm names used by TaskBuilder.

Web UI name	TaskBuilder active variable name
PA	**PA**
PA not detected	alarm-pa-not-detected
Firmware invalid	alarm-pa-firmware-invalid
Calibration invalid	alarm-pa-calibration-invalid
Forward power low	alarm-pa-forward-power-low
Power foldback	alarm-pa-power-foldback
Reverse power high	alarm-pa-reverse-power-high
Shutdown	alarm-pa-shutdown
VSWR high	alarm-pa-vswr-high
Driver current high	alarm-pa-driver-current-high
Final 1 current high	alarm-pa-final1-current-high
Final 2 current high	alarm-pa-final2-current-high
Current imbalance	alarm-pa-current-imbalance
Supply voltage low	alarm-pa-supply-voltage-low
Supply voltage high	alarm-pa-supply-voltage-high
Driver temperature high	alarm-pa-driver-temperature-high
Final 1 temperature high	alarm-pa-final1-temperature-high
Final 2 temperature high	alarm-pa-final2-temperature-high
PMU	**PMU**
PMU not detected	alarm-pmu-not-detected
Firmware invalid	alarm-pmu-firmware-invalid
Mains supply failed	alarm-pmu-mains-supply-failed
Power up fault	alarm-pmu-power-up-fault
Shutdown imminent	alarm-pmu-shutdown-imminent
Temperature high	alarm-pmu-temperature-high
Battery protection mode	alarm-pmu-battery-protection-mode
Battery voltage low	alarm-pmu-battery-voltage-low
Battery voltage high	alarm-pmu-battery-voltage-high
Output current high	alarm-pmu-output-current-high
Output voltage low	alarm-pmu-output-voltage-low
Output voltage high	alarm-pmu-output-voltage-high

Web UI name	TaskBuilder active variable name
System	**System**
Ambient temperature low	alarm-ambient-temperature-low
Ambient temperature high	alarm-ambient-temperature-high
External reference absent	alarm-external-reference-absent
1PPS pulse absent	alarm-1pps-pulse-absent
QoS jitter	alarm-qos-jitter
QoS lost packets	alarm-qos-lost-packets
Transmit buffer	alarm-transmit-buffer
Fallback controlled	alarm-fallback-controlled
Duplicate node priority	alarm-duplicate-node-priority
NTP unsynchronized	alarm-ntp-unsynchronized
Site synchronization unaligned	alarm-site-synchronization-unaligned
TxR cable absent	alarm-txr-cable-absent
Cartesian loop unstable	alarm-cartesian-loop-unstable
Reciter	**Reciter**
Channel invalid	alarm-channel-invalid
Temperature high	alarm-reciter-temperature-high
Simulcast unsynchronized	alarm-simulcast-unsynchronized
Transmitter calibration invalid	alarm-transmitter-calibration-invalid
Receiver calibration invalid	alarm-receiver-calibration-invalid
Hardware configuration invalid	alarm-hardware-configuration-invalid
25 MHz synthesizer out of lock	alarm-25-mhz-synthesizer-out-of-lock
61.44 MHz synthesizer out of lock	alarm-61-44-mhz-synthesizer-out-of-lock
TxF synthesizer out of lock	alarm-txf-synthesizer-out-of-lock
TxR synthesizer out of lock	alarm-txr-synthesizer-out-of-lock
Rx synthesizer out of lock	alarm-rx-synthesizer-out-of-lock
Receiver unsynchronized	alarm-receiver-unsynchronized

Web UI name	TaskBuilder active variable name
Custom	**Custom**
CUSTOM - Alarm 1	alarm-custom-alarm-1
CUSTOM - Alarm 2	alarm-custom-alarm-2
CUSTOM - Alarm 3	alarm-custom-alarm-3
CUSTOM - Alarm 4	alarm-custom-alarm-4
CUSTOM - Alarm 5	alarm-custom-alarm-5
CUSTOM - Alarm 6	alarm-custom-alarm-6
CUSTOM - Alarm 7	alarm-custom-alarm-7
CUSTOM - Alarm 8	alarm-custom-alarm-8
CUSTOM - Alarm 9	alarm-custom-alarm-9
CUSTOM - Alarm 10	alarm-custom-alarm-10
CUSTOM - Alarm 11	alarm-custom-alarm-11
CUSTOM - Alarm 12	alarm-custom-alarm-12
Front panel	**Front panel**
Fan 1	alarm-fan-1
Fan 2	alarm-fan-2
Fan 3	alarm-fan-3
FP not detected	alarm-front-panel-not-detected
Invalid firmware	alarm-front-panel-invalid-firmware

Referencing an alarm that is not defined on that platform means that the TaskBuilder program will fail with a parse error. Examples are:

- alarm-system-site-synchronization-unaligned with DMR firmware
- alarm-front-panel-not-detected with TB7300 base station.

12 Specifications and Limits

Parameter	Value	Description
Event response latency	100 ms nominal	The time that TaskBuilder may take to respond to an event. May increase if base station is heavily loaded.
Throughput	50 events per second nominal	Number of rules that can be triggered. May decrease if base station is heavily loaded.
Maximum out-standing events	20	Events may be discarded if there are more pending events than the value listed.

13 Change History

Release 3.25

`dig-out-13` is an output-only digital pin with the ability to sink current for driving a relay.

`trace:` statement allows programs to write directly to the log.

Web-UI: [Undo] button allows you to revert to the last good TaskBuilder program

Release 3.20

TaskBuilder is intended for general release.

Leading zeroes

In base station release 3.20, some of the standard variable names have been changed to remove leading zeroes, for example;

- `channel.023` becomes `channel.23`,
- `dig-in-03` becomes `dig-in-3`,
- `dig-out-03` becomes `dig-out-3`.

Case sensitivity

Names are case-insensitive. In release 3.20, names are converted internally to lower case, and displayed (Web UI, logs) in lower case. In future releases, names will still be case-insensitive, but the original case of the input text will be preserved.

TaskBuilder keywords are case-insensitive. They are displayed in lower case independently of the case in the input text.

TaskBuilder exit event

TaskBuilder can respond to exit events as well as start up events. On TaskBuilder exit, TaskBuilder will execute the actions for rules that include a `when: operation.stopping` clause.

TaskBuilder control over custom alarms

TaskBuilder program actions can now raise or clear base station custom alarms.