Andrew Gellnick, UTMC Officer, Balfour Beatty Living Places, Buckinghamshire Highways Alliance

Utilising the GO bit to manage critical internal queuing in a SCOOT network.

As traffic control engineers, we can all think of a junction where a link presents a challenge in managing queuing to prevent blocking back. A typical example would be an internal link in a signal-controlled roundabout or gyratory. Other situations could be where a yellow box junction or 'Keep Clear' marking have been painted in an attempt to keep intersections clear of queues that we observe from cycle to cycle. Sometimes, these queues are caused by interruptions to flow between nodes that we have no control over and perhaps all we can do as a precaution is to hold extra traffic back every cycle, reducing the overall capacity.

In the past, we might have used the hurry call facility for Vehicle Actuated junctions using a call-cancel detector, but as we migrate increasingly more junctions to UTC SCOOT there is a need to be able to replicate this within SCOOT. There are facilities within SCOOT for reacting to queuing such as 'Gating', but they result in gradual and incremental adjustments, and can be difficult to fine tune. The ability to generate a hurry call stage change within SCOOT is a very useful facility for the SCOOT control engineer's 'toolbox'.

This paper and the presentation that will accompany it describes how the controller code can be specially conditioned so that the "Gap Out bit" (also known as the "GO bit") can be repurposed to generate an immediate change in stage when a queue is detected. This represents a net gain in overall facility because for its originally intended purpose the GO bit is underutilised.

It is based on the SWARCO controller code and has been possible because of the open nature of the controller coding and ease at which changes in configurations can be undertaken and emulated with minimal manufacturer involvement. However, the result should be applicable to other manufacturers' controllers as well.

Original purpose of the "Gap-Out bit"

The Gap Out (GO) bit is used to advance a stage change when extensions expire for the current stage, so that SCOOT gives a degree of autonomy to the local controller to make a stage change decision, without generating a SCOOT error. This could be for early termination of a traffic stage or advancing a pedestrian stage when traffic demand gaps out. In practise, within SCOOT regions these situations do not often arise because SCOOT tends to regulate traffic consistently. For links near the periphery of SCOOT regions, the facility is probably more beneficial.

The GO bit is very controllable because it is activated on the UTC Interface at a specific point in the SCOOT stage translation plan by appending the mnemonic "+GO" to the Force bits, meaning it can be introduced or omitted simply by changing plan, and the position within the cycle at which it becomes active can be varied via the offset. It appears and disappears during the cycle similar to a demand dependent 'window'. By way of a fictious example:

{F1 0}S1, {F2 0, F1F2 2}S2 - SCOOT translation plan with a 2 second window for Demand dependent stage 2

{F1 0}S1, {F2 0, F1F2 2, **F1F2+GO** 20}S2 – adding the GO bit for an early move back to stage 1 when extensions for stage 2 expire.

Controller manufacturers have implemented the Gap Out facility differently. The SWARCO controller achieves its stage changes by the interaction between code statements that represent all **demands** for a stage and all the factors that **inhibit** the stage change until the appropriate instant. Each stage-to-stage movement, for each mode of control, has its own pair of code statements; one for '**demand**' and one for '**inhibit'**. They are initiated in the code as defaults which can become overridden by subsequent statements during the configuration process.

The SWARCO default code includes a term for the GO bit, so for the originally intended use of the GO bit, the code is already there and the GO bit can be employed simply by including the GO bit on the controller UTC interface and O.T.U configuration and then adding "+GO" to the SCOOT plan.

SWARCO controller code for a UTC stage t demand; (t=stage 2 in this case):

(1) dUTCd2(t) = uF(t) || (uFD(t) && (uD(t) || dr(B) || dr(C)));

Decoded:

A demand for stage 2 exists if there is a force bit for the stage OR a demand dependent force bit combined with a demand; which can either be by D bit for the stage or on street VA detector demands, in this case (for a particular installation), phases B and C run in stage 2.

Default SWARCO controller code for inhibiting a UTC stage change from stage f to stage t; (from stage 1 in this case):

(2) dUTCi1(f,t) = { [uF(f) || uFD(f)] && [(uGO(xp)==0) || ex(t)] };

Note: Curly and Square brackets substituted to make the code easier to decipher; f = from stage, t = to stage, xp = stream number, u = UTC flag, F=Force bit, FD=DD Force bit, ex(t) = extension timer running for a phase that would terminate for the next stage.

Decoded:

In most situations there is no Gap Out bit configured on the UTC interface, so the half of statement (2) to the right of the AND (&&) is always logical 1 (TRUE) and the effective code reduces to statement (3) below:

(3) dUTCi1(f,t) = ((uF(f) || uFD(f)); i.e. the 'inhibit move' is active to hold the stage until all the Force bits for the stage are off, and this accords with our experience watching UTC running.

However, when the Gap Out bit is configured, the whole expression (2) needs to be true for the inhibit to hold the current stage by preventing the stage change.

When the inhibit is dropped (OFF/FALSE/Logical 0), the controller can move to the next demanded stage. Interpreting this code statement is more difficult as we naturally think of code doing something when it is 'ON', but here the opposite is true; the Inhibit being ON ensures no change, in other words, it acts like a brake.

So, to maintain the Inhibit and keep the current stage running, both code segments either side of the '&&' for statement (2) need to be TRUE.

Cancelling the inhibit to permit a stage change occurs when:

1) the F or FD force bits drop, or
2) the F or FD force bits are still being received but a Gap Out bit is set (logical 1) and no VA extensions are running.

This makes sense just thinking logically, but I invite you to reach the same conclusion from the original equation (2) using De Morgan's Theorem.

Alternatively, consider that with the GO set to Logical 1, the term (uGO(xp)==0) is FALSE and the expression needs the VA extensions being logical 1,active, to keep the inhibit ON.

The GO bit is an underused powerful feature because in essence, it allows SCOOT to defer a stage change decision to local conditions at a specific point in the SCOOT plan, without SCOOT generating an error. Alternatively, as in most cases, it can be excluded simply by omitting the GO command from the SCOOT plan. So why not make use of it in another way?

N.B: At the time of writing, the following idea has not yet been implemented on site due to delays with communications, but it has been tested successfully in the SWARCO controller emulator. I foresee no reason why it won't work because SCOOT does not know the signal controller logic code; it just knows it is in a stage, and if there is a GO bit active, the controller might move autonomously to that next stage in advance of the SCOOT plan telling it to do so. I hope that by the time of the conference I will be able to confirm this works as intended on-street.

Repurposing the "GO bit" for a queue controlling Hurry Call

Detecting a queue is easy; using a detector and the call/cancel facility of the controller applied to a dummy phase. The trigger sensitivity can be adjusted in RAM by altering the call and cancel delays, so an appropriate pair of values can be fine-tuned.

The next step is to assign the dummy phase to the stage that you would like the controller to move to when a queue is detected. This might be a contingency stage intended just to clear queues, and maybe it only appears in UTC mode.

The final step is to modify the controller inhibit code for the relevant stage to stage movements, replacing the VA extensions term with the (inverted) demand for the dummy phase as demonstrated next with the aid of colour coding. Note that once this is done, the original Gap Out facility will not work for that stage-to-stage change, but you probably were not using it anyway!

Referring back to the default code statement (2) reproduced below, the difference is only in the extension term which is replaced with an inverted phase demand term. The inversion makes sense as now we are looking for a demand rather than the end of it. Statement 4 below is the generic default code statement modified for this queue control idea.

```
(2) dUTCi1(f,t) = ((uF(f) || uFD(f))  &&  ((uGO(xp)==0) || ex(t)));
(4) dUTCi1(f,t) = ((uF(f) || uFD(f))  &&  ((uGO(xp)==0) || !(dr(t)));
```

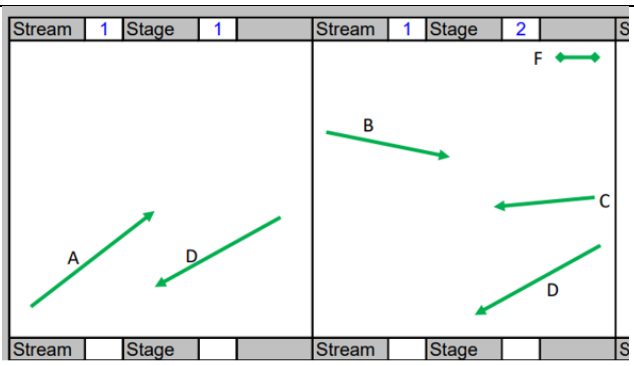
Note: ! = NOT, i.e. inverted

dr(t) is a controller demand reply for a phase in the 'to' stage, in this case a demand for the dummy phase we have assigned to the call/cancel loop.

The above statements are in SWARCO's generic code form, hence the 'd' suffix meaning 'default'.

When modified to match the specific requirements of the site, the code takes on a modified 'm' format to override the preceding default. So, for a stage move 1 to 2, where the queuing detection dummy phase is phase F running in stage 2, the specific code for the inhibit might be as in statement 5 below:

```
(5) mUTC1_2i() = in(UTC_I1) && ((uGO(0)==0) || (dr(F)==0));
```



- UTC_I1 in this case is the allocation of bit 1 of the inputs to UTC flag F1.
- uGO(0) is the GO bit for the first stream
- dr(F)==0 is an inverted demand for phase F which is the dummy phase set up to detect queuing,
- This code allows a stage 1 to 2 move when Queuing is detected (phase F) with GO bit set even if F1 is still set.

The SCOOT stage translation plan may look like this example:

{F1 0, F1F2+GO 20}S1, {F2 0, F1F2 2}S2

So, during SCOOT stage S1, when UTC stage F1 is running, at the point 20 seconds after the start of SCOOT stage S1, a queue detected on the dummy phase F combined with the GO bit set will cause the controller to move early to UTC stage F2 whilst SCOOT stage S1 is still active.

The point in the cycle at which the presence of a queue is considered can be moved around the plan by varying the event time; set at 20 seconds from the start of SCOOT stage 2 in this hypothetical example.

Recall that this is for the SWARCO controller but hopefully other controller manufacturers will be able to mimic this.

That is all there is to it. Now we have a tool to use within SCOOT that could alleviate the need for box junction markings in some locations.

This is a simplification of my first implementation of a more complicated design which has a choice of stages to move to due to queuing in two directions. In this case, this requires an extra contingency stage that is only used in UTC, and adjustment to the demand code statement to override the stage order so that a 1-3-2 sequence is possible when a demand window for F2 and F3 is running. The purpose of this first paper is to demonstrate the principles as simply as possible.

If you are interested to see the proof for the Inhibit code changes, or even just the process of developing code from first principles, read on. It is good practise to develop and specify Special Conditioning this way as it eliminates ambiguity that usually occurs when writing it in plain English so that we have a better chance of getting it right first time and without unforeseen consequences.

From First Principles using Boolean Algebra

Each line is a set of 3 binary variables that occur together, in logic, they are AND statements. 8 lines covers all combinations of 3 variables, each providing a result of 1 or 0, (TRUE or FALSE), these are linked together as logic OR statements.

Events that cancel the inhibit: F1 inactive or (F1 active AND GO bit active AND Queue detected).

ANDs →

ORs ↓

F1 a	GO active b	Queue detected c	Resulting Inhibit f
1	0	0	1; TRUE
1	0	1	1; TRUE
1	1	0	1; TRUE
1	1	1	0
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0

Normally the F bit active holds the stage by keeping the Inhibit ON. The exceptional condition is highlighted in yellow where:

F1 && GO && (Queue detected) → NOT Inhibit.

We need to develop an expression that gives the correct result for all combinations of input. The first 3 rows give a result of TRUE for the inhibit,(hold stage) and the remaining 5 rows give the result FALSE allowing the stage change.

Looking at the 3 TRUE results we create a BOOLEAN expression with 3 terms, using column sub-headers a,b,c and f as simple substitutions for the notation, and then apply the Boolean laws to simplify it.

a.NOTb.NOTc + a.NOTb.c + a.b.NOTc = f [Starting expression from the lines when f = TRUE]

a.NOTb.(NOTc + c) + a.b.NOTc = f [Distributive law, OR form]

a.NOTb.(1) + a.b.NOTc = f [Complement Law]

a.NOTb + a.b.NOTc = f [Identity Law]

a.(NOTb + (b.NOTc)) = f [Distributive law, OR form]

a.(NOTb + b).(NOTb + NOTc) = f

a.(1).(NOTb + NOTc) = f [Complement Law]

a.(NOTb + NOTc) = f [Identity Law]

[Absorption law]

Substituting back for a, b, c and adjusting the notation type:

F1 AND ($\overline{\text{GO}}$ OR $\overline{\text{Queue detected}}$) = Resulting Inhibit; and this is correct for all the 8 rows in the table above.

Evidently, the Inhibit is OFF while the F bit is still present if both GO and Queue detector are ON.

This accords with statement (4) and the modified code example statement (5):

(4) dUTCi1(f,t) = ((uF(f) || uFD(f)) && ((uGO(xp)==0) || !(dr(t)));

(5) mUTC1_2i() = in(UTC_I1) && ((uGO(0)==0) || (dr(F)==0));